



AFRL-RI-RS-TR-2018-108

GENERAL PURPOSE PROBABILISTIC PROGRAMMING PLATFORM WITH EFFECTIVE STOCHASTIC INFERENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

APRIL 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2018-108 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

WILLIAM E. MCKEEVER JR.
Work Unit Manager

/ S /

JOHN D. MATYJAS
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
<small>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</small> PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) APRIL 2018		2. REPORT TYPE FINAL TECHNICAL REPORT		3. DATES COVERED (From - To) NOV 2013 – DEC 2017	
4. TITLE AND SUBTITLE GENERAL PURPOSE PROBABILISTIC PROGRAMMING PLATFORM WITH EFFECTIVE STOCHASTIC INFERENCE				5a. CONTRACT NUMBER FA8750-14-2-0004	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) Vikash Mansinghka				5d. PROJECT NUMBER PPML	
				5e. TASK NUMBER 2M	
				5f. WORK UNIT NUMBER IT	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology (MIT) 77 Massachusetts Avenue Cambridge MA 02139-4301				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-108	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Probabilistic modeling and machine learning have proven to be powerful tools in many defense, industrial, and scientific computing applications. Unfortunately, their continuing adoption has been hindered because engineering with them requires PhD-level expertise. Our research in this program led to the creation of multiple open-source probabilistic programming languages. These languages achieved key program goals, such as (i) reducing the lines of code required to build state-of-the-art machine learning systems by ~50x; (ii) making machine learning and data science capabilities accessible to a broader class of programmers, by providing automatic model discovery mechanisms and simple, SQL-like query languages; (iii) making it possible to deploy rich generative models to solve applied problems, and thereby solve hard 3D computer vision problems with no training data; and (iv) revealing interfaces and abstractions that unify a broad set of probabilistic programming languages and enable multiple inference strategies or "solvers" to interoperate.					
15. SUBJECT TERMS Probabilistic programming, programming languages					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 87	19a. NAME OF RESPONSIBLE PERSON WILLIAM E. MCKEEVER JR.
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (315) 330-2897

1.0	SUMMARY	1
2.0	INTRODUCTION	3
2.1	An example of probabilistic programming	5
2.2	Venture	10
2.3	BayesDB	12
2.4	Picture	17
2.5	MetaProb	20
3.0	METHODS, ASSUMPTIONS, AND PROCEDURES	22
4.0	RESULTS AND DISCUSSION	23
4.1	Results for solving inference problems using custom inference strategies written in Venture .	23
4.2	Example probabilistic programs and meta-programs written in Metaprob.	35
4.3	Example results drawn from program challenge problems	50
4.3.1	January 2017 Hackathon - Gapminder Dataset	50
4.3.2	Challenge Problem #7 - Flu Problem 2017	53
4.4	Example results for our team challenge problem of analyzing a database of Earth satellites using BayesDB.	56
4.5	Example results showing how probabilistic programming can be used to reimplement the ``Automatic Statistician'' in under 70 lines of code.	59
4.6	Example results showing how probabilistic programming can be used to solve problems of visual scene understanding.	65
5.0	CONCLUSION	73
6.0	REFERENCES	74
	LIST OF ACRONYMS	80

List of Figures

Figure 1. The problem of inferring curves from data while simultaneously choosing the functional form of the curve and rejecting outliers.	5
Figure 2. An example probabilistic model for solving the problem of inferring curves that explain data and rejecting outliers.	6
Figure 3. Probabilistic program source code for solving the same problem (left), along with an example execution trace of this probabilistic program (right).	7
Figure 4. Additional example execution traces of the probabilistic program from the previous figure.	7
Figure 5. Probabilistic source code for an extended model that includes stochastic choices for the model hyper-parameters.	8
Figure 6. An overview of the interface to inference in probabilistic programs.	9
Figure 7. An example of querying the probabilistic program shown earlier given a single observed datapoint.	9
Figure 8. Querying a probabilistic program given several observed datapoints. Note that the generated curves all roughly align with the observed data.	10
Figure 9. Choosing the right inference strategy for the problem (Mansinghka, Selsam and Perov, 2014; Mansinghka, Schaechtle, Radul, Handa, Rinard, in review).	11
Figure 10. Data science is difficult – Adapted from (Drew Conway, 2013). This figure summarizes the problems that BayesDB is designed to solve.	13
Figure 11. BayesDB: probabilistic programming with Bayesian model discovery. BayesDB is applicable to data science problems that range along the spectrum of model detail, from descriptive and exploratory analysis to predictive, causal, and mechanistic modeling.	13
Figure 12. Example datasets to which BayesDB has been applied.	14
Figure 13. An overview of the architecture of BayesDB.	14
Figure 14. An example of Bayesian model discovery in BayesDB applied to a database of Senate voting records.	15
Figure 15. A mathematical description non-parametric Bayesian prior used for Bayesian model discovery in BayesDB.	15
Figure 16. Example queries in BayesDB against two probabilistic programs that can be generated by the non-parametric Bayesian prior from the previous figure.	16
Figure 17. Computer vision (bottom path) as the inverse problem to computer graphics (top path).	18
Figure 18. An illustration of generative probabilistic graphics for 3D road finding. On the left, results from Aly et al (2006). On the right, typical inference results from the proposed generative probabilistic graphics approach.	19
Figure 19. Four input images from our CAPTCHA corpus, along with the final results and convergence trajectory of typical inference runs.	20
Figure 20. Gaussian Process Structure Learning using an inference strategy that applies Metropolis-Hastings to subproblems.	25
Figure 21. Gaussian Process Structure Learning with a different inference strategy.	26
Figure 22. Venture model code for the Gaussian process structure learning program. This probabilistic model code defines a probabilistic model over structured Gaussian process models for time series data.	27
Figure 23. Venture observation and inference code for the Gaussian process structure learning program. This code shows different custom inference strategies for discovering Gaussian process model structures.	28
Figure 24. A graphical model representing the variables and dependencies for our second benchmark problem: a stochastic volatility model.	29
Figure 25. Stochastic Volatility Model with two types of observation sequences and three inference operators: single-site Metropolis-Hasting operating on individual random choices, global Metropolis-Hastings for all random choices, and particle Gibbs.	29

Figure 26. Shows Venture code for the stochastic volatility model, along with code for custom inference.	30
Figure 27. A third benchmark problem: logistic regression for classifying handwritten digits from the MNIST database.	31
Figure 28. Venture code for the Bayesian logistic regression example from the previous figure	31
Figure 29. Venture code for a more complex Dirichlet process mixture of experts model, extending the previous logistic regression benchmark.	32
Figure 30. Accuracy results for the Dirichlet process mixture of experts.	33
Figure 31. Venture model component, observation component, and inference components for a bipartite Bayesian network inference problem.	34
Figure 32. A comparison of the accuracy of custom inference strategies on the bipartite Bayesian network	35
Figure 33. A Metaprob transcript, modeling n flips of a potentially biased coin.	39
Figure 34. Two traces of executing the flip_coins program from Figure 1 for two flips.	40
Figure 35. Definition of the classic earthquake-burglary Bayes net in Metaprob, including a function for collecting the variables of interest from a trace of its execution. (Radul and Mansinghka, 2018)	41
Figure 36. The joint distribution induced by intervening on the earthquake-burglary Bayes net to set the alarm variable to true.	42
Figure 37. Basic operations on Metaprob traces, showing the Metaprob cod and results.	43
Figure 38. An illustration of trace prefix selection. Each dashed box highlights which subtrace is shown in the subsequent pane.	44
Figure 39. A user-space meta-program for performing a single step of “lightweight” Metropolis-Hastings inference.	45
Figure 40. Markov-chain Monte Carlo inference over the user-space Gaussian distribution.	46
Figure 41. Metaprob source code for a program that generates data from a Dirichlet process mixture of Gaussians with randomly chosen hyper-parameters.	47
Figure 42. Metaprob source code for a meta-program that adds an observed dataset, does 1000 steps of lightweight Metropolis-Hastings inference on executions of the program from Figure 33, and then generates 100 samples from the predictive distribution.	48
Figure 43. Progress of inference in the Dirichlet process mixture of Gaussians model with hyperparameter inference from Figure 42, above.	49
Figure 44. Influenza-like illness rates in Tennessee, including both raw data and forecasts from the probabilistic programs developed for this challenge problem.	54
Figure 45. Influenza-like illness rates in Texas, including both raw data and forecasted rates from the probabilistic program developed to solve this challenge problem.	54
Figure 46. Hierarchical Bayesian model for the joint distribution of N dependent time series $\{x_n\}$	55
Figure 47. Quantitative evaluation of forecasting performance using several simple and state-of-the-art baselines	56
Figure 48. Example application: Earth-orbiting satellites.	57
Figure 49. Example probabilistic code written in BayesDB for Bayesian model discovery from this satellites database.	57
Figure 50. Example probabilistic code written in BayesDB for improving over baseline Bayesian model discovery by integrating custom models	58
Figure 51. An example query in BayesDB’s Bayesian Query Language (BQL) that identifies probably anomalous satellites, along with example results.	58
Figure 52. Query performance results on the Satellites database, showing improvement in runtime for multiple queries between our first submission and second submission.	59
Figure 53. The passenger counts can be decomposed into a linearly increasing trend, a periodic overlay (with amplitude that increases over time), and a set of local stochastic deviations from the main trend.	60
Figure 54. Executing synthesized model programs to produce Gaussian process datasets.	62

Figure 55. Synthesis model: AST prior G - Structure discovery in time series via probabilistic program synthesis.	62
Figure 56. Source code for the Automatic Statistician in Venture	63
Figure 57. Qualitative results for extrapolation and interpolation tasks.	63
Figure 58. Samples from prior for Airline example.	64
Figure 59. Samples for approximate posterior for Airline example.	64
Figure 60. Our probabilistic program implements functionality from the “Automated Statistician” but using ~100x fewer lines of code. (Schaechtle et al., 2017).	65
Figure 61. Picture code [left] for the 3D face application,, along with a schematic description of the dependencies within that Picture program [right]. (Kulkarni et al., 2015).	67
Figure 62. An overview of the inference architecture from Picture, in which learned bottom-up proposals can be combined with custom search operators. (Kulkarni et al., 2015).	68
Figure 63. Quantitative and qualitative results for 3D human pose program.	68
Figure 64. Bottom: Random program traces sampled from the prior during training.	70
Figure 65. A detailed illustration of the utility of learned bottom-up proposals for inference in Picture programs.	70
Figure 66. random samples	71
Figure 67. Picture inference on representative faces, answering the question "what does a given face probably look like when rotated or lit differently?"	71

1.0 SUMMARY

Probabilistic modeling and machine learning have proven to be powerful tools in many defense, industrial, and scientific computing applications. Unfortunately, their continuing adoption has been hindered because engineering with them requires PhD-level expertise. Building these systems involves simultaneously developing mathematical models, inference algorithms and optimized software implementations. Small changes to the underlying modeling assumptions, data, or compute budget often require end-to-end redesigns and re-implementations that can lead to surprising changes to system accuracy and performance.

DARPA's Probabilistic Programming for Advancing Machine Learning (PPAML) program aimed to address these problems. Specific goals identified by the program were (i) reducing the lines of code required to build state-of-the-art machine learning systems; (ii) making machine learning and data science capabilities accessible to a broader class of programmers; (iii) making it possible to deploy rich generative models to solve applied problems, and thereby reduce the amount of training data required, and (iv) identifying interfaces and abstractions that unify probabilistic programming languages and enable multiple inference strategies or "solvers" to interoperate. The program also aimed to create the technical foundations for new collaborations between the machine learning and programming language research communities, so that future progress could be made by incremental research and engineering.

Over the course of the program, we succeeded in achieving these ambitious goals. Specific highlights include:

1. We developed Venture, a probabilistic programming system with support for custom inference strategies. We showed that Venture makes it possible to reimplement and extend the Automatic Statistician, a state-of-the-art machine learning system, in just 70 lines of code, or roughly ~50X fewer lines of code than the original implementation.
2. We developed BayesDB, which makes it possible for ordinary developers to solve hard data science and machine learning problems using a simple, SQL-like language. BayesDB was evaluated by DARPA during PPAML via two hackathons, showing that it is possible to use probabilistic programming to solve data science problems with an accuracy that exceeds solutions produced by expert statistical consultants. BayesDB has yielded multiple industry and government partnerships and been adopted as a component of the DARPA Synergistic Discovery and Design (SD2) program.
3. We developed Picture, which makes it possible to write ~50 line programs that use generative modeling to solve hard 3D computer vision problems. These programs require no training data at all, but match or exceed the accuracy of state-of-the-art baselines that are based on machine learning from training data. Picture won a best paper (honorable mention) award at CVPR 2015, the top international computer vision conference.
4. We developed Metaprob, a probabilistic meta-programming language. In Metaprob, both probabilistic models and custom inference strategies can be written as user-space code in

a single language, using novel reflective programming constructs. It is thus sufficiently expressive that Venture, Picture, and BayesDB --- as well as earlier languages such as Church --- could (in principle) all be re-implemented within it. It also addresses a core intellectual challenge posed by Dr. Kathleen Fisher, the original PPAML program manager: how to integrate approaches to inference in probabilistic programs.

Our PPAML research generated significant interest and investment from industry. For example, it led to invited briefings for the CEOs of Intel Corporation (given by PI Vikash Mansinghka) and Microsoft Corporation (given by Co-PI Josh Tenenbaum). It also led to the formation of Empirical Systems, a VC-backed startup based on BayesDB, and the deployment of the BayesDB open-source prototype at JP Morgan. Engineers at other companies, including Zymergen, Inc., have used BayesDB to detect predictive relationships between variables.

Our work on PPAML has also begun to catalyze the creation of an academic community in probabilistic programming. For example, PIs Mansinghka and Tenenbaum gave an invited tutorial at NIPS 2017 on probabilistic programming, with over 2000 attendees. Mansinghka also gave invited tutorials at multiple O'Reilly AI conferences, with hundreds of participants. This work also led to the first full-semester graduate course in probabilistic programming, taught at MIT by PI Vikash Mansinghka, and a book contract for an introductory book on Probabilistic Programming with the MIT Press. Preparation for all these tutorials, courses, and teaching material was supported by PPAML.

This report is structured as follows. The next section gives an introduction to probabilistic programming and a brief overview of the main probabilistic languages we developed over the course of the program. The methods section outlines the research approach. The results and discussion section gives representative quantitative and qualitative results showcasing capabilities of PPAML technology. Finally, the conclusion outlines recommendations that could guide future research investments in probabilistic programming and highlights steps being taken to transition probabilistic programming to problems of pressing interest to the US government, especially the Department of Defense.

2.0 INTRODUCTION¹

Probabilistic modeling and machine learning have proven to be powerful tools in many defense, industrial, and scientific computing applications. Unfortunately, their continuing adoption has been hindered because engineering with them requires PhD-level expertise. Building these systems involves simultaneously developing mathematical models, inference algorithms and optimized software implementations. Small changes to the underlying modeling assumptions, data, or compute budget often require end-to-end redesigns and reimplementations that can lead to surprising changes to system accuracy and performance.

Probabilistic programming is an emerging field at the intersection of artificial intelligence and programming languages, that aims to address these problems. By encapsulating and automating inference, learning and prediction, probabilistic programming provides a promising path forwards, and has the potential to radically simplify the process of building systems that use probabilistic representations of knowledge to interpret empirical data. But after DARPA's investment in PPAML, what is probabilistic programming known to be useful for, in practice? and what are the fundamental technical problems that organize research in the field?

Probabilistic programming is already influencing research in several other fields. The most visible successes are currently in Bayesian data analysis, where probabilistic programming has made state-of-the-art hierarchical modeling and Monte Carlo inference techniques accessible to a broad audience. For example, the Stan language for hierarchical Bayesian modeling has been adopted by a community of thousands of scientists and statisticians. Probabilistic programming languages also make it possible to rapidly prototype systems based on inference in probabilistic models that would otherwise be prohibitively complex. Example applications include detecting and localizing nuclear explosions on the earth's surface, inferring 3D object geometry from single images, and determining the probable lineages of malware. Probabilistic programming languages have been used to reimplement state-of-the-art techniques for AI-assisted data analysis in ~50x fewer lines of code than is possible in Python or MATLAB. Finally, probabilistic programs are the basis of new models of human-level concept learning and reasoning. For example, a recent model of how people can learn handwritten characters from just one training example is formulated in terms of probabilistic program induction, i.e. learning probabilistic programs from data.

Probabilistic programming is based on two technical ideas:

Probabilistic models can be represented using programs. These programs can generate samples from the probability distribution associated with the model, or calculate the probability density of values sampled from the distribution.

Operations on models such as inference and learning can be implemented as meta-programs. These meta-programs produce, consume, execute, transform, or modify other probabilistic programs. Their inputs and outputs are other probabilistic programs, represented as

¹ This section is derived from Mansinghka et al., 2015; Kulkarni et al., 2015; Mansinghka, Selsam, and Perov, 2014.

executables and/or source code, as well as representations of other probabilistic programs' runtime behavior.

These two ideas lead to a broad design space of languages and engineering approaches supported by those languages. Some probabilistic languages are embedded in widely-used programming languages; for example, Figaro is embedded in Scala, WebPPL is embedded in JavaScript, Anglican is embedded in Clojure, and Picture is embedded in Julia. Other probabilistic languages are stand-alone; prominent examples include BLOG, Stan, and Venture. Some languages, such as Church, BLOG, and Stan, are based on black-box inference engines, while others, such as Venture, Picture, and Edward, provide constructs that let users specify custom inference strategies. Some languages and implementations are closely designed around the needs of a specific class of applications, while others aim to be more general-purpose. Some languages have explored the value of restricting expressivity to ensure that specific inference strategies apply to all programs. For example, Infer.NET does not support models defined in terms of recursive processes, which enables static compilation into graphical models. No one approach appears to be best suited for all problem domains.

Attempts to formalize these ideas have also yielded new theorems that connect fundamental concepts in probability theory with theoretical computer science. For example, what are the limits of mechanizing probabilistic inference? Ackerman, Freer, and Roy (2010) recently showed that it is possible to construct a pair of computable random variables (X, Y) in the unit interval whose conditional distribution $P[Y|X]$ encodes the halting problem. However, they also showed that inference can be mechanized as long as measurements are known to be sufficiently noisy. These theorems show that studying probabilistic objects and operations in computational terms can lead to new insights. They also provide important constraints on the design of probabilistic programming languages.

At the start of the PPAML program, the available languages and systems were largely unsuitable for use by practitioners. Examples of drawbacks include inadequate and unpredictable runtime performance, limited expressiveness, batch-only operation, lack of extensibility, and reliance on opaque approximation techniques for inference. The first aim of our research in PPAML was to develop technology that enabled users to reliably write, test, debug, and optimize probabilistic programs and reason about their behavior, without requiring detailed knowledge about how inference is implemented. A second aim of our research was to demonstrate the effectiveness of probabilistic programming on a broad class of important problems.

We initially proposed to focus on developing Venture, an interactive platform for general-purpose probabilistic programming that delivers practical and predictable inference performance. However, over the course of the program it became clear different programming languages were needed to target different domains, specially the "data science" use cases emphasized by DARPA's PPAML challenge problems, and problems of visual scene understanding that are of broad interest to the defense community. It also became clear that there were fundamental opportunities to do research in probabilistic meta-programming, motivated by capability limitations in Venture that were highlighted by DARPA's challenge problems. This report describes both the research we initially proposed and the evolution of the research over the course of the program. It thus outlines the development of Venture, but also BayesDB, Picture,

and MetaProb. Together, these languages informed our solutions to DARPA's challenge problems, to the teaching of probabilistic programming at industry conferences, and to the solution of the team challenge problems we described in our initial proposal.

2.1 An example of probabilistic programming

Consider the problem of inferring curves from noisy data. The figure below gives an example of this problem, along with example results from three probabilistic models that each attempt to solve the problem.

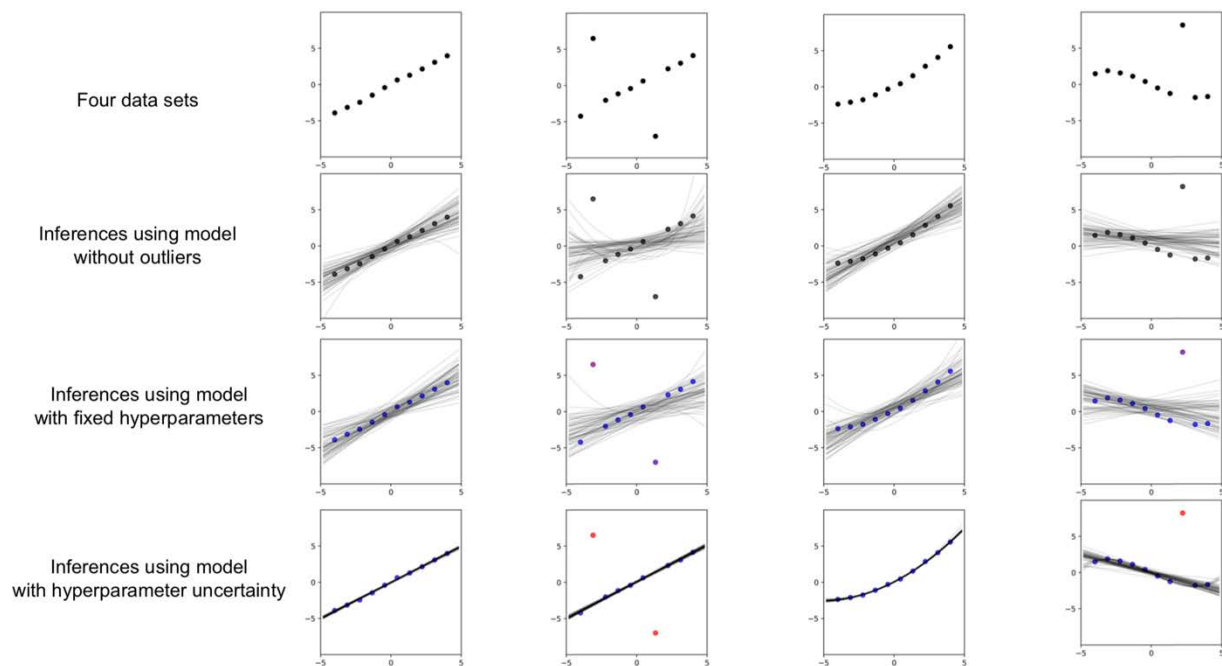


Figure 1. The problem of inferring curves from data while simultaneously choosing the functional form of the curve and rejecting outliers. The top row shows example datasets, while the next three rows show inference results from three probabilistic models. Inference results include the data (with dots shaded according to the probability with which they are an outlier) and randomly sampled curves that explain the data (drawn in grey). The first model has no random variables encoding outliers. The second model supports outliers but makes fixed assumptions about a broad class of model hyperparameters. The third model (bottom row) allows those hyperparameters to be inferred from the data. It is the only model that can robustly solve all of the example problems.

$$\begin{aligned}
k &\sim \text{Uniform}(\{1, 2, 3, 4\}) \\
\boldsymbol{\theta} &\sim \text{Normal}(\mathbf{0}_{k+1}, \mathbf{I}_{k+1}) \\
z_i &\sim \text{Bernoulli}(0.1) \text{ for } i = 1 \dots N \\
y_i &\sim \begin{cases} \text{Normal}(\sum_{j=1}^{k+1} x_i^{j-1} \theta_j, 1) & \text{if } z_i = 0 \\ \text{Normal}(\sum_{j=1}^{k+1} x_i^{j-1} \theta_j, 10) & \text{if } z_i = 1 \end{cases} \text{ for } i = 1 \dots N
\end{aligned}$$

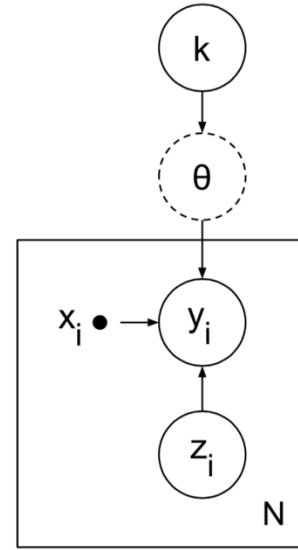


Figure 2. An example probabilistic model for solving the problem of inferring curves that explain data and rejecting outliers. The left column shows the random variables in standard mathematical notation. The right column shows an equivalent probabilistic graphical model.

This type of probabilistic model includes several characteristics that make it difficult to solve. First, the effective number of parameters in the model is itself a random variable. Second, there is one binary variable for each datapoint, determining whether or not the datapoint is modeled by the underlying curve or is treated as an outlier.

However, this type of model is straightforward to represent as a probabilistic program. Probabilistic programs represent models using code that makes stochastic choices for each latent variable in the model. This approach allows powerful programming constructs to be used to define probabilistic modeling assumptions. For example, conditional statements can be used to encode choices over model structure. The next figure gives an example probabilistic program encoding this model, along with an example execution trace of this program.

```

@probabilistic function model(x::Vector{Float64})

# prior over degree of polynomial
degree_prior = [0.25, 0.25, 0.25, 0.25]

# generate degree (either 1, 2, 3, or 4)
degree = @choice(categorical(degree_prior), "degree")

# generate parameters
parameters = Vector{Float64}(degree+1)
for k=1:(degree+1)
    parameters[k] = @choice(normal(prior_mean, prior_std), "theta-$k")
end

# generate data
y = Vector{Float64}(length(x))
for i=1:length(x)
    if degree == 1
        y_mean = dot(parameters, [1., x[i]])
    elseif degree == 2
        y_mean = dot(parameters, [1., x[i], x[i]^2])
    elseif degree == 3
        y_mean = dot(parameters, [1., x[i], x[i]^2, x[i]^3])
    else
        y_mean = dot(parameters, [1., x[i], x[i]^2, x[i]^3, x[i]^4])
    end
    is_outlier = @choice(flip(prob_outlier), "outlier-$i")
    noise = is_outlier ? outlier_noise : inlier_noise
    y[i] = @choice(normal(y_mean, noise), "y-$i")
end
end

```

As a probabilistic program

"degree"	1
"theta-1"	1.20
"theta-2"	-0.20
"outlier-1"	false
"outlier-2"	false
"outlier-3"	false
"outlier-4"	false
"y-1"	-0.22
"y-2"	0.10
"y-3"	-0.70
"y-4"	1.60

One possible **execution trace** of the program

with input $x = [-3, 0, 2, 3]$
and output $y = [-0.22, 0.1, -0.70, 1.60]$

Figure 3. Probabilistic program source code for solving the same problem (left), along with an example execution trace of this probabilistic program (right).

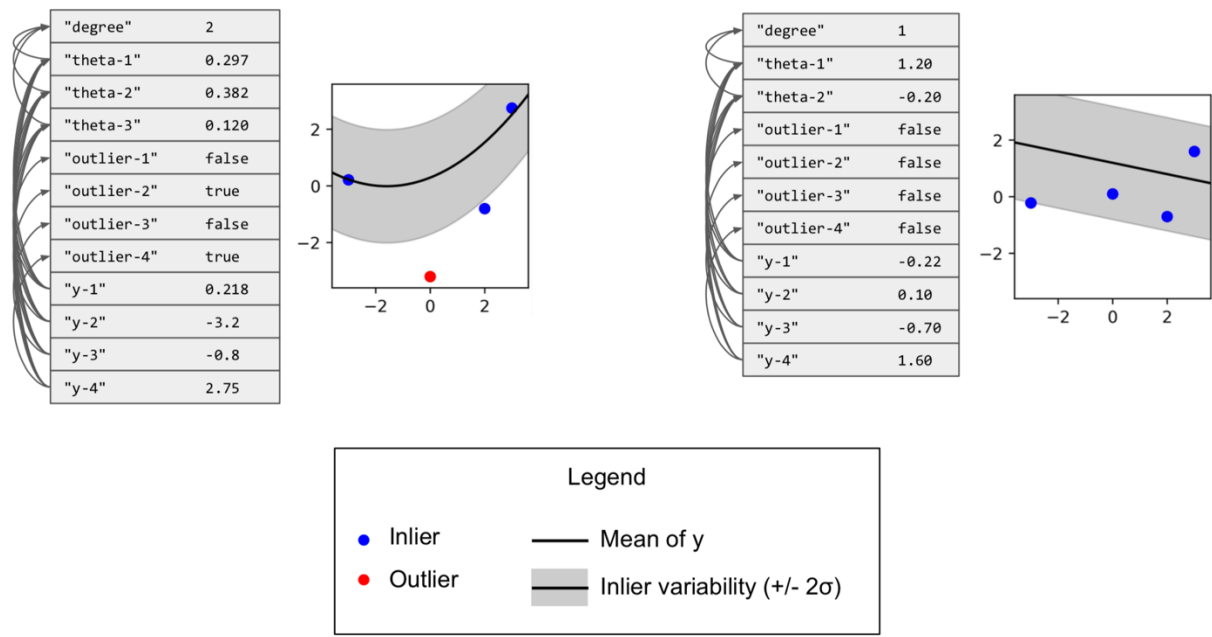


Figure 4. Additional example execution traces of the probabilistic program from the previous figure.

```

@probabilistic function model(x::Vector{Float64})

    # prior over degree of polynomial
    degree_prior = [0.25, 0.25, 0.25, 0.25]

    # generate degree (either 1, 2, 3, or 4)
    degree = @choice(categorical(degree_prior), "degree")

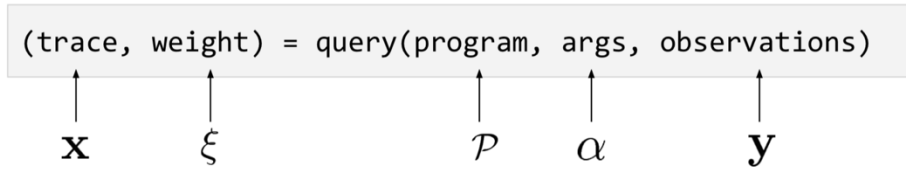
    # generate parameters
    parameters = Vector{Float64}(degree+1)
    for k=1:(degree+1)
        parameters[k] = @choice(normal(0, 1), "theta-$k")
    end

    # hyperparameters
    inlier_noise = @choice(gamma(2., 1.), "inlier-noise")
    outlier_noise = @choice(gamma(10., 1.), "outlier-noise")
    prob_outlier = @choice(beta(1., 20.), "prob-outlier")

    # generate data
    y = Vector{Float64}(length(x))
    for i=1:length(x)
        if degree == 1
            y_mean = dot(parameters, [1., x[i]])
        elseif degree == 2
            y_mean = dot(parameters, [1., x[i], x[i]^2])
        elseif degree == 3
            y_mean = dot(parameters, [1., x[i], x[i]^2, x[i]^3])
        else
            y_mean = dot(parameters, [1., x[i], x[i]^2, x[i]^3, x[i]^4])
        end
        is_outlier = @choice(flip(prob_outlier), "outlier-$i")
        noise = is_outlier ? outlier_noise : inlier_noise
        y[i] = @choice(normal(y_mean, noise), "y-$i")
    end
end

```

Figure 5. Probabilistic source code for an extended model that includes stochastic choices for the model hyper-parameters.



Distribution on traces induced by executing program $p(\mathbf{x}; \mathcal{P}, \alpha)$
(e.g. the prior)

Distribution on traces conditioned on observations $p(\mathbf{x}|\mathbf{y}; \mathcal{P}, \alpha) \propto p(\mathbf{x}; \mathcal{P}, \alpha) \prod_{i \in \mathbf{y}} \delta(x_i, y_i)$
(e.g. the posterior)

Distribution on traces sampled during query execution $q(\mathbf{x}; \mathcal{P}, \alpha, \mathbf{y}) \approx p(\mathbf{x}|\mathbf{y}; \mathcal{P}, \alpha)$
(e.g. the posterior approximation)

Figure 6. An overview of the interface to inference in probabilistic programs.

Probabilistic programming languages provide mechanisms for inferring the probable execution traces of programs given data. The data provides constraints on the choices in the execution traces of the probabilistic code. The above figure gives a schematic description of this notion of inference.

```

observations = Trace()
observations["y-2"] = 0.0
(trace, weight) = query(model, ([-3, 0, 2, 3],), observations)

```

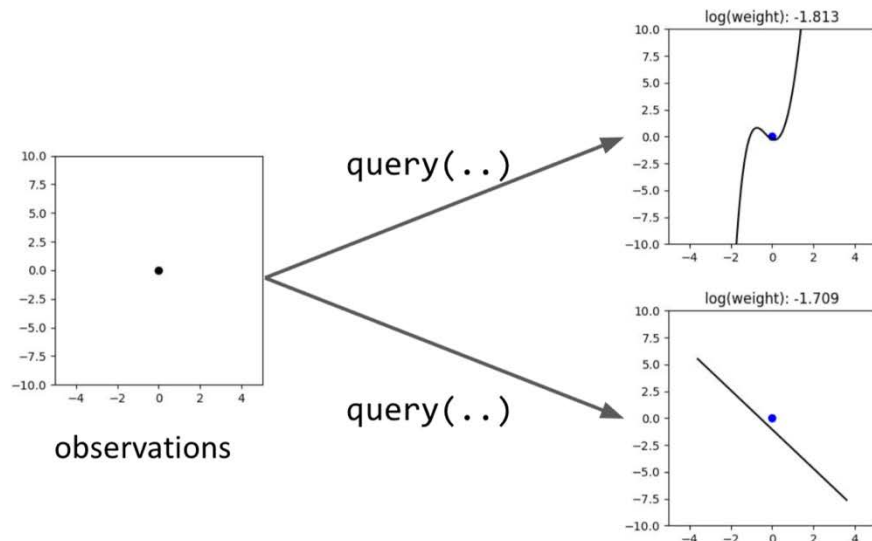


Figure 7. An example of querying the probabilistic program shown earlier given a single observed datapoint. Given this one data point, a broad class of curves of very different shapes are all probable.

```

observations = Trace()
observations["y-1"] = -3.0
observations["y-2"] = 0.0
observations["y-3"] = 2.0
observations["y-4"] = 3.0
(trace, weight) = query(model, ([-3, 0, 2, 3],), observations)

```

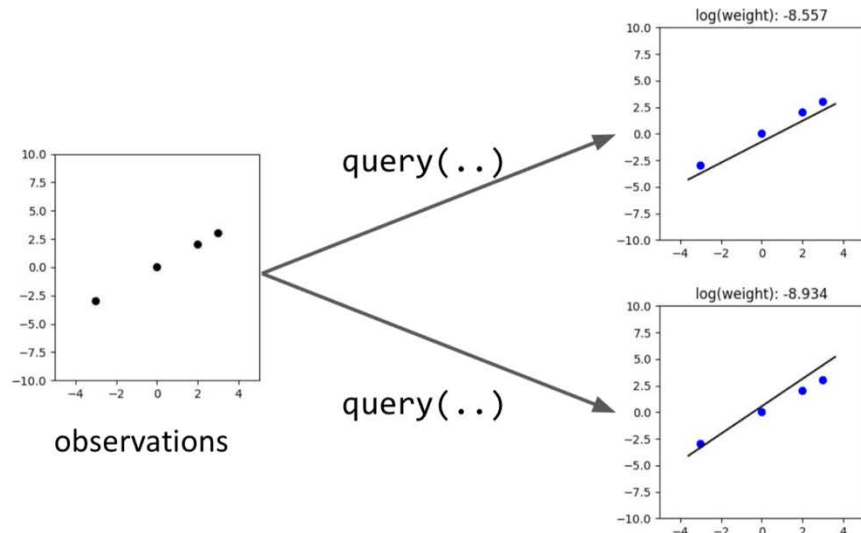


Figure 8. Querying a probabilistic program given several observed datapoints. Note that the generated curves all roughly align with the observed data.

2.2 Venture

Venture descends from Church, one of the first Turing-universal, higher-order probabilistic programming languages — introduced and developed by a research group which included several members of our team. Venture retains the expressiveness of the Church language, while adding interactivity, greater extensibility, and vastly improved efficiency, scalability, and predictability.

The specific focus of our work on Venture was supporting the use of custom inference strategies. Custom inference strategies to solve probabilistic modeling and inference problems is now mainstream practice across a broad range of fields. Current probabilistic programming languages, however, typically provide only a specific black-box inference strategy or set of strategies. Over the course of the PPAML program, we developed novel probabilistic programming constructs for custom inference strategies. Below, we describe results obtained by implementing these strategies in the Venture probabilistic programming language. These constructs enable developers to identify inference sub-problems, then apply an appropriately chosen inference tactic to each identified sub-problem. Experimental results on a collection of probabilistic programs, written in the Venture probabilistic programming language, highlight the significant performance and accuracy benefits that custom inference strategies can deliver.

Our goals with Venture over the course of the program were to:

1. Reduce the development time and code complexity needed to build a wide range of state-of-the-art machine learning and probabilistic modeling systems.
2. Enable domain experts to solve typical data analysis problems in their domains using short probabilistic scripts and custom variations on standard modeling templates, without deep technical expertise in machine learning or probabilistic inference.
3. Enable probabilistic programming experts to prototype and incrementally optimize machine learning systems that are currently infeasible to build at all.

In doing this, we have substantially advanced the technical foundations of probabilistic programming and produced an open-source system that demonstrates the utility of these advances. Venture is now significantly more efficient and extensible than previous implementations, and Venture programs are simpler to understand, test, debug and optimize. We have improved Venture’s scalability, inference latency, and usability.

It is easy to find examples that demonstrate that custom inference is important in practice. For example, the following figure shows that even on the simple problem of inferring curves from data, with outliers, the choice of inference strategy matters. Different inference strategies (left) yield significantly different inferences (right). In these figures, the inferred curves are shown in blue, real data in green, and true outliers in red. Note that for all but the strategy on the top left, the curves do not consistently line up with the real data.

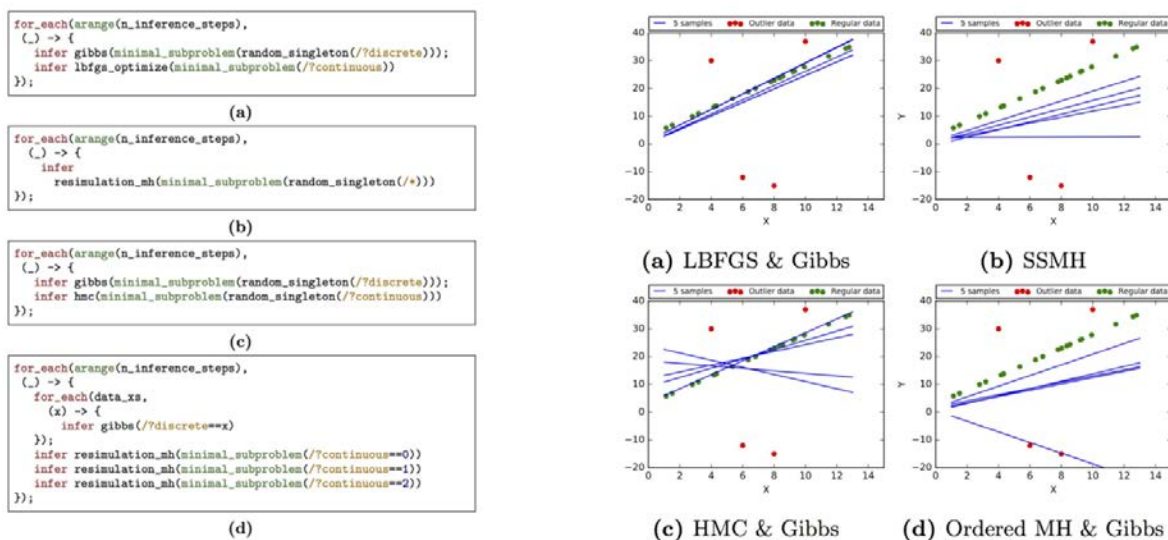


Figure 9. Choosing the right inference strategy for the problem (Mansinghka, Selsam and Perov, 2014[1]; Mansinghka, Schaehtle, Radul, Handa, Rinard, in review).

2.3 BayesDB

Another probabilistic programming language we developed over the course of PPAML was BayesDB. BayesDB was developed in response to the demand for applications to data science problems and performed well in multiple DARPA evaluations during PPAML. BayesDB is a platform for AI-assisted data science that enables domain experts to answer questions in seconds or minutes that otherwise require hours or days of work by someone with good statistical judgment.

Why is this important? Stakeholders in business, humanitarian work, science, and government are increasingly recognizing the importance of making statistical inferences from their data. Existing approaches to this problem require experts in statistical modeling or data scientists proficient in applied machine learning. These skills are projected to be in short supply as the importance of statistical inference is increasingly recognized across a variety of fields. Also, developers new to machine learning may be stymied by the maze that is the current machine learning toolkit. This toolkit can come up short in settings that don't match canonical machine learning problems.

How does BayesDB solve these problems? First, BayesDB provides a simple, SQL-like query language for asking data science queries. This language can be used for data search, inferential statistics, and predictive modeling applications. Second, BayesDB provides AI assistance for exploratory data analysis and baseline statistical modeling via CrossCat, a probabilistic method that emulates many of the judgment calls ordinarily made by a human data analyst. This AI assistance enables domain experts who lack training in statistics to draw rigorous inferences from messy real-world databases. Third, BayesDB provides an advanced "meta-modeling" language for customizing the AI's modeling assumptions to incorporate both quantitative and qualitative domain knowledge. This enables expert statisticians to improve inference quality in risk-sensitive applications.

Example questions that the BayesDB open-source prototype has been used to answer include the following:

1. Probabilistic search: query by example ("find me the 10 colleges most like MIT and Harvard with regards to graduates' median income, but admissions rates over 10%"), and query by probability ("find me colleges with the most unexpectedly high mean income for graduates").
2. AI assessment of data quality: "find me radiation measurements from the Safecast.org project that are most likely to be errors and/or legitimate anomalies."
3. Virtual experiments: "generate some EEG measurements we might expect for a child in Bangladesh at age 3 years of age, given all the other EEG measurements we have observed so far."
4. AI-assisted inferential statistics: "what genetic markers, if any, predict increased risk of suicide given a PTSD diagnosis? and how confident can we be in the amount of increase, given uncertainty due to statistical sampling and the large number of possible alternative explanations?"

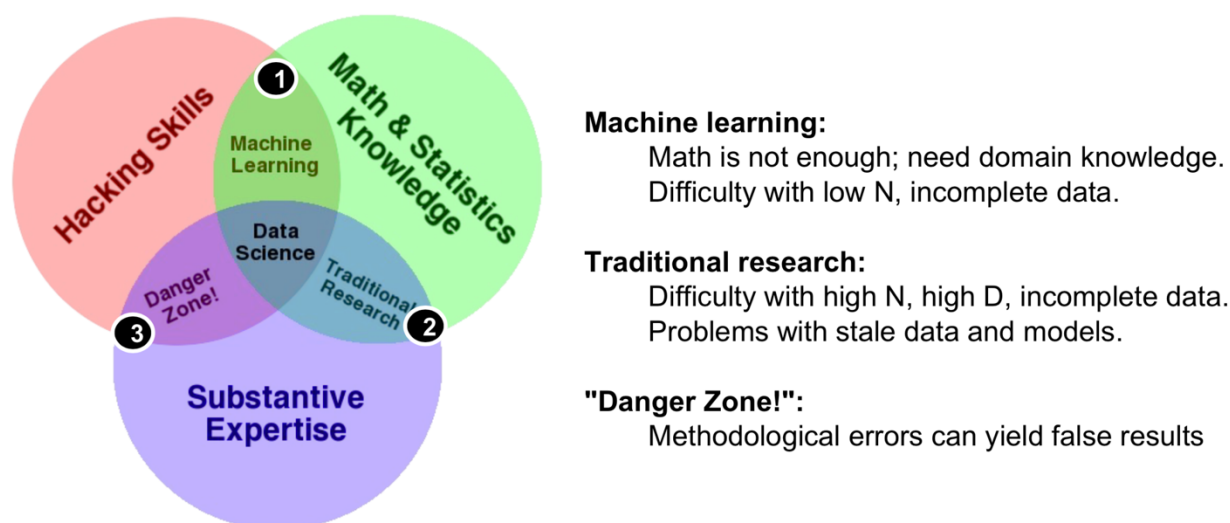


Figure 10. Data science is difficult – Adapted from (Drew Conway, 2013). This figure summarizes the problems that BayesDB is designed to solve.

BayesDB can enable people in the "danger zone" to rely on AI assistance in lieu of math & statistics knowledge. BayesDB's probabilistic programming language for Bayesian model discovery also addresses technical limitations of standard techniques in machine learning and statistics. It thus has the potential to be useful to people in the "machine learning" and "traditional research" sections of the diagram.

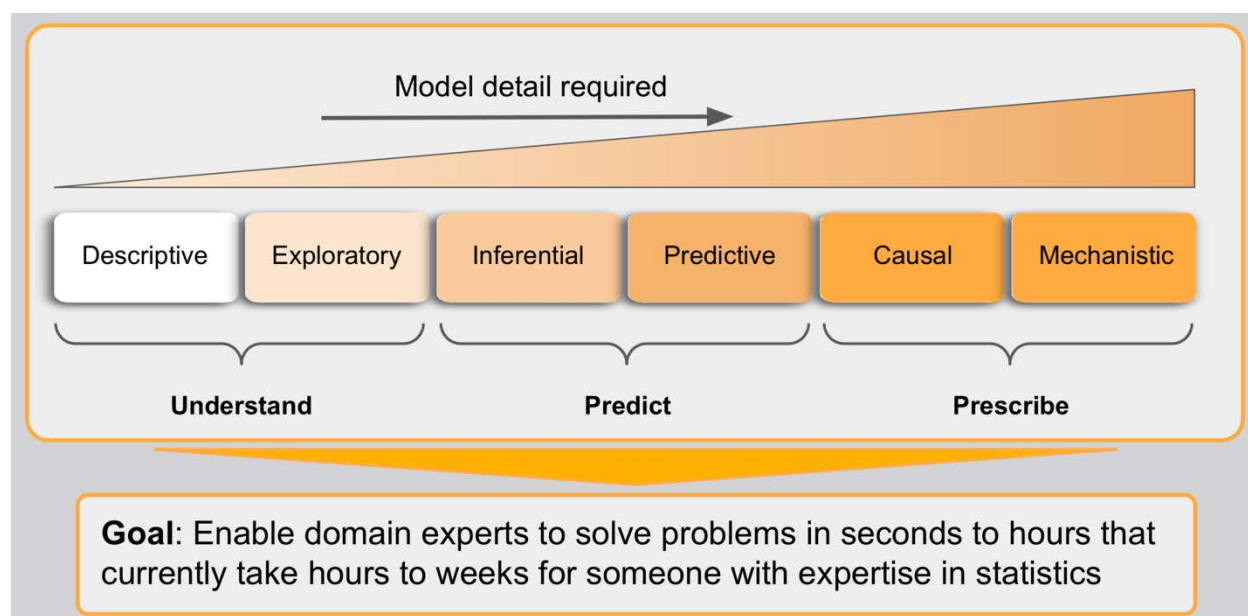


Figure 11. BayesDB: probabilistic programming with Bayesian model discovery. BayesDB is applicable to data science problems that range along the spectrum of model detail, from descriptive and exploratory analysis to predictive, causal, and mechanistic modeling.

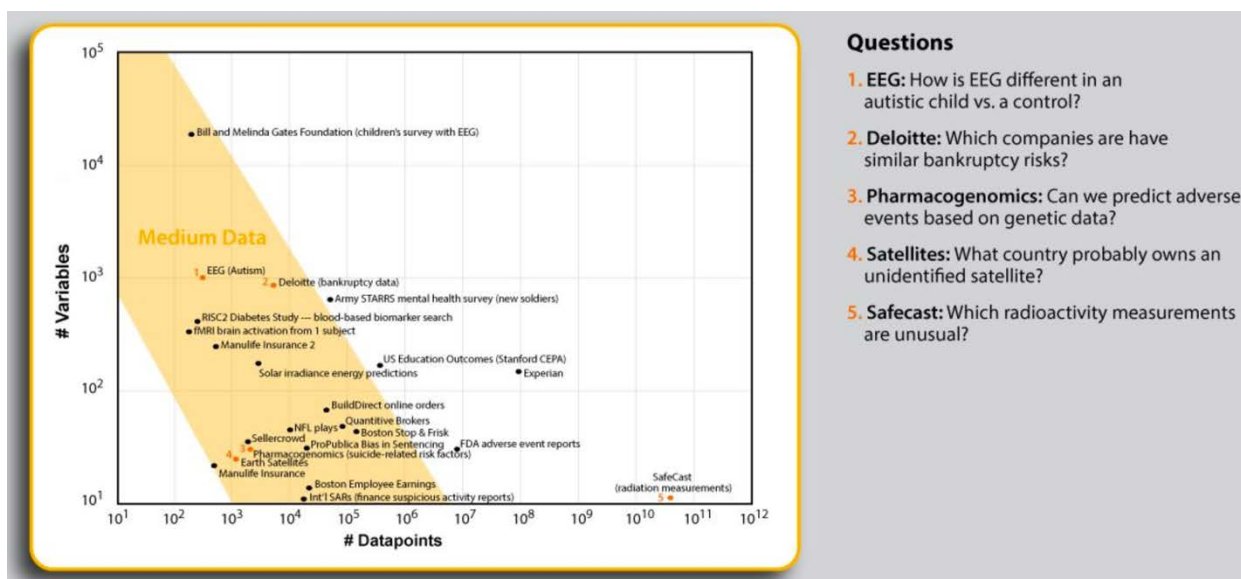


Figure 12. Example datasets to which BayesDB has been applied.

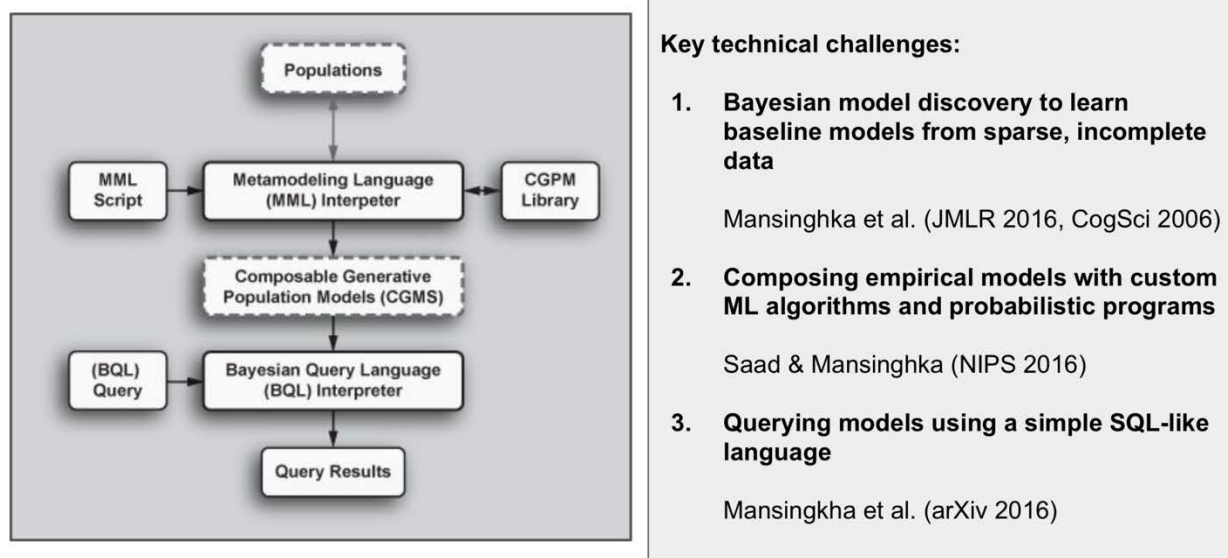


Figure 13. An overview of the architecture of BayesDB.

Building BayesDB required solving technical challenges such as (i) automatically discovering models from sparse, incomplete data; (ii) combining empirical models with custom ML algorithms and probabilistic programs; and (iii) providing a simple, SQL-like language by which end users can query these models.

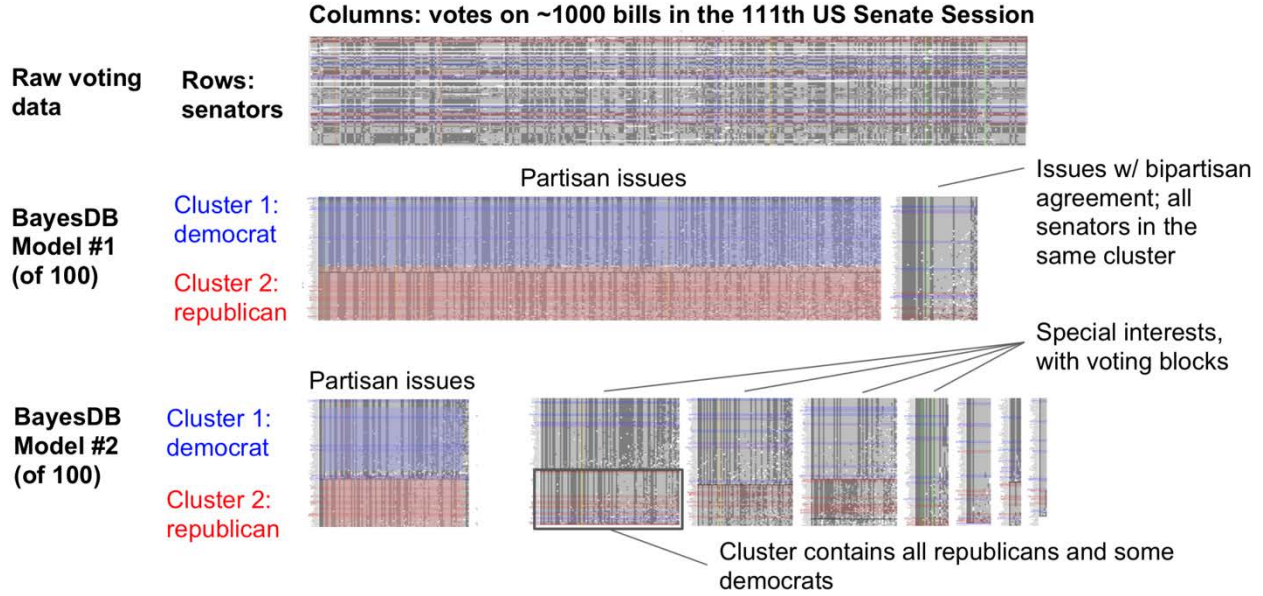


Figure 14. An example of Bayesian model discovery in BayesDB applied to a database of Senate voting records.

BayesDB can discover models from a broad class of real world databases. The above figure shows an example of the structure of the models discovered by BayesDB from voting records for the 111th US Senate (2009). Some senators are highlighted in colors based on their generally accepted identification as democrats (blue), moderates (purple), or republicans (red). The middle row shows a simple or low resolution posterior sample that divides bills into those that exhibit partisan alignment and those with bipartisan agreement. Clusters of senators, generated automatically, are separated by thick black horizontal lines. The bottom row shows a sample that includes additional views and clusters, positing a finer-grained predictive model for votes that are treated as random in the middle row. (Mansinghka et al., 2016).

$$\begin{aligned}
 \alpha_D &\sim \text{Gamma}(k=1, \theta=1) \\
 \vec{\lambda}_d &\sim V_d(\cdot) \\
 z_d &\sim \text{CRP}(\{z_i \mid i \neq d\}; \alpha_D) \\
 \alpha_v &\sim \text{Gamma}(k=1, \theta=1) \\
 y_r^v &\sim \text{CRP}(\{y_i^v \mid i \neq r\}; \alpha_v) \\
 \vec{\theta}_c^d &\sim M_d(\cdot; \vec{\lambda}_d) \\
 \vec{x}_{(\cdot, d)}^c &= \{x_{(r, d)} \mid y_r^{z_d} = c\} \sim \begin{cases} \prod_r L_d(\vec{\theta}_c^d) & \text{if } u_d = 1 \\ ML_d(\vec{\lambda}_d) & \text{if } u_d = 0 \end{cases}
 \end{aligned}$$

$$\begin{aligned}
 &\text{foreach } d \in \{1, \dots, D\} \\
 &\text{foreach } d \in \{1, \dots, D\} \\
 &\text{foreach } v \in \vec{z} \\
 &\text{foreach } v \in \vec{z} \text{ and} \\
 &\quad r \in \{1, \dots, R\} \\
 &\text{foreach } v \in \vec{z}, c \in \vec{y}^v, \text{ and } d \text{ such that} \\
 &\quad z_d = v \text{ and } u_d = 1 \\
 &\text{foreach } v \in \vec{z} \text{ and each } c \in \vec{y}^v
 \end{aligned}$$

Figure 15. A mathematical description non-parametric Bayesian prior used for Bayesian model discovery in BayesDB. This probabilistic generative model induces a prior distribution over a hypothesis space of probabilistic programs that model multivariate data. (Mansinghka et al., 2016)

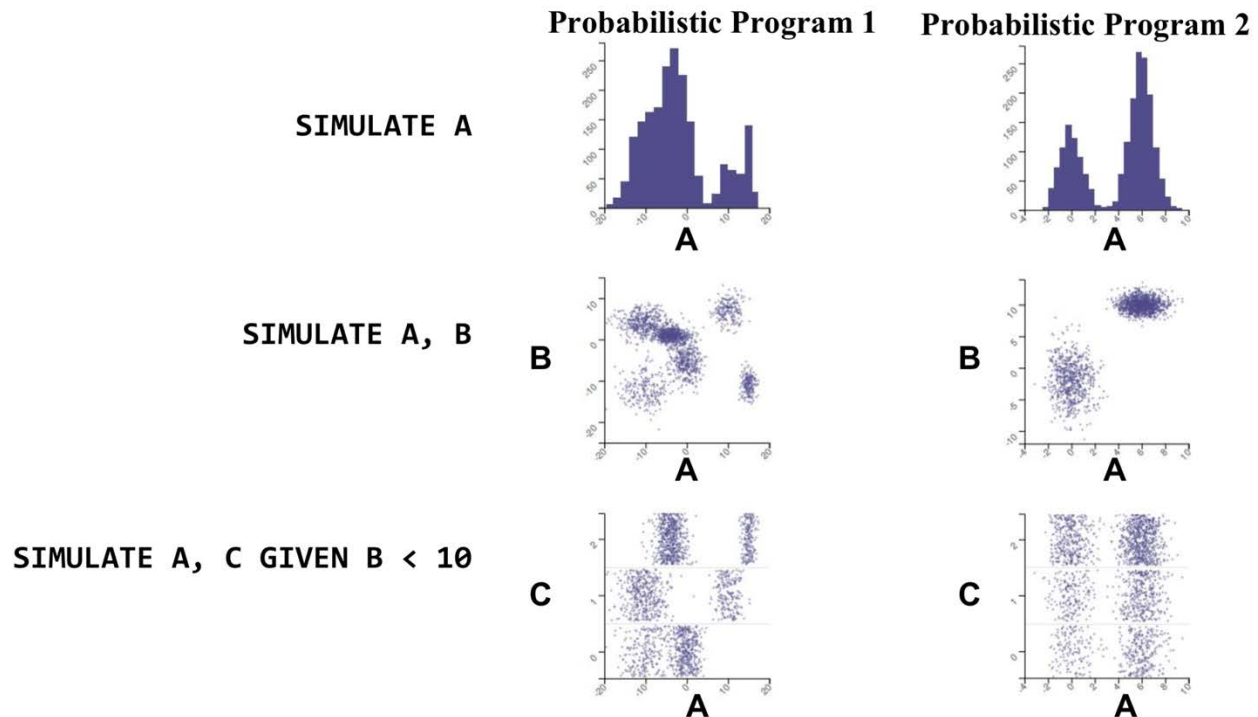


Figure 16. Example queries in BayesDB against two probabilistic programs that can be generated by the non-parametric Bayesian prior from the previous figure.

From a technical perspective, BayesDB consists of four components, integrated into a single probabilistic programming system:

1. The Bayesian Query Language (BQL), an SQL-like query language for Bayesian data analysis. BQL programs can solve a broad class of data analysis problems using statistically rigorous formulations of cleaning, exploration, confirmatory analysis, and predictive modeling. BQL defines these primitive operations for these workflows in terms of Bayesian model averaging over results from an implicit set of multivariate probabilistic models.
2. A mathematical interface that enables a broad class of multivariate probabilistic models, called generative population models, to be used to implement BQL. According to this interface, a data generating process defined over a fixed set of variables is represented by (i) an infinite array of random realizations of the process, including any observed data, and (ii) algorithms for simulating from arbitrary conditional distributions and calculating arbitrary conditional densities. This interface permits many statistical operations to be implemented once, independent of the specific models that will be used to apply these operations in the context of a particular data table.
3. The BayesDB Meta-modeling Language (MML), a minimal probabilistic programming language. MML includes constructs that enable statisticians to integrate

custom statistical models — including arbitrary algorithmic models contained in external software — with the output of a broad class of Bayesian model building techniques. MML also includes constructs for specifying qualitative dependence and independence constraints.

4. A hierarchical, semi-parametric Bayesian “meta-model” that automatically builds ensembles of generalized mixture models from database tables. These ensembles serve as baseline data generators that BQL can use for data cleaning, initial exploration, and other routine applications.

This design insulates end users from most statistical considerations. Queries are posed in a qualitative probabilistic programming language for Bayesian data analysis that hides the details of probabilistic modeling and inference. Baseline models can be built automatically and customized by statisticians when necessary. All models can be critically assessed and qualitatively validated via predictive checks that compare synthetic rows (generated via BQL’s SIMULATE operation) with rows from the original data. Instead of hypothesis testing, dependencies between variables are obtained via Bayesian model selection.

BayesDB is “Bayesian” in two ways:

1. In BQL, the objects of inference are rows, and the underlying probability model forms a “prior” probability distribution on the fields of these rows. This is then constrained by row-specific observations to create a posterior distribution of field values. Without this prior, it would be impossible to simulate rows or infer missing values from partial observations.

2. In MML, the default meta-model is Bayesian in that it assigns a prior probability to a very broad class of probabilistic models and narrows down on probable models via Bayesian inference. This prior is unusual in that it encodes a state of ignorance rather than a strong inductive constraint. MML also provides instructions for augmenting this prior to incorporate qualitative and quantitative domain knowledge.

In practice, it is useful to use BQL for Bayesian queries against models built using non-Bayesian or only partially Bayesian techniques. For example, MML supports composing the default meta-model with modeling techniques specified in external code that need not be Bayesian. However, the default is to be Bayesian for both model building and query interpretation, as this ensures the broadest applicability of the results.

2.4 Picture

The idea of computer vision as the Bayesian inverse problem to computer graphics has a long history and an appealing elegance, but it has proved difficult to directly implement. Instead, most vision tasks are approached via complex bottom-up processing pipelines. Our research has shown that it is possible to write short, simple probabilistic graphics programs that define flexible generative models and to automatically invert them to interpret real-world images. It has also shown that custom inference strategies can improve performance on solving these problems. If scaled up, this approach has the potential to address a broad class of visual scene

understanding problems faced by the Department of Defense, and could yield solutions with both reduced code complexity, improved accuracy, and other functional benefits such as the ability to report uncertainty in scene percepts for use by downstream processing.

Picture is a domain specific probabilistic programming language for this class of problems. The approach implemented by Picture is called *generative probabilistic graphics programming*. Generative probabilistic graphics programs consist of a stochastic scene generator, a renderer based on graphics software, a stochastic likelihood model linking the renderer's output and the data, and latent variables that adjust the fidelity of the renderer and the tolerance of the likelihood model. Representations and algorithms from computer graphics, originally designed to produce high-quality images, are instead used as the deterministic backbone for highly approximate and stochastic generative models. This formulation combines probabilistic programming, computer graphics, and approximate Bayesian computation, and depends only on general-purpose, automatic inference techniques.

Example applications of this approach include: (i) reading sequences of degraded and adversarially obscured alphanumeric characters; (ii) inferring 3D road models from vehicle-mounted camera images; (iii) inferring 3D models of faces from single images, and rendering those models in different viewpoints and lighting conditions; (iv) inferring models of 3D human bodies from single images, including challenging images from outdoor scenes and bodies that are partially occluded; and (v) inferring 3D models of radially symmetric objects such as cups and lamps and bottles from household images. Each of the probabilistic graphics programs we present relies on under 50 lines of probabilistic code, and supports accurate, approximately Bayesian inferences about ambiguous real-world images.

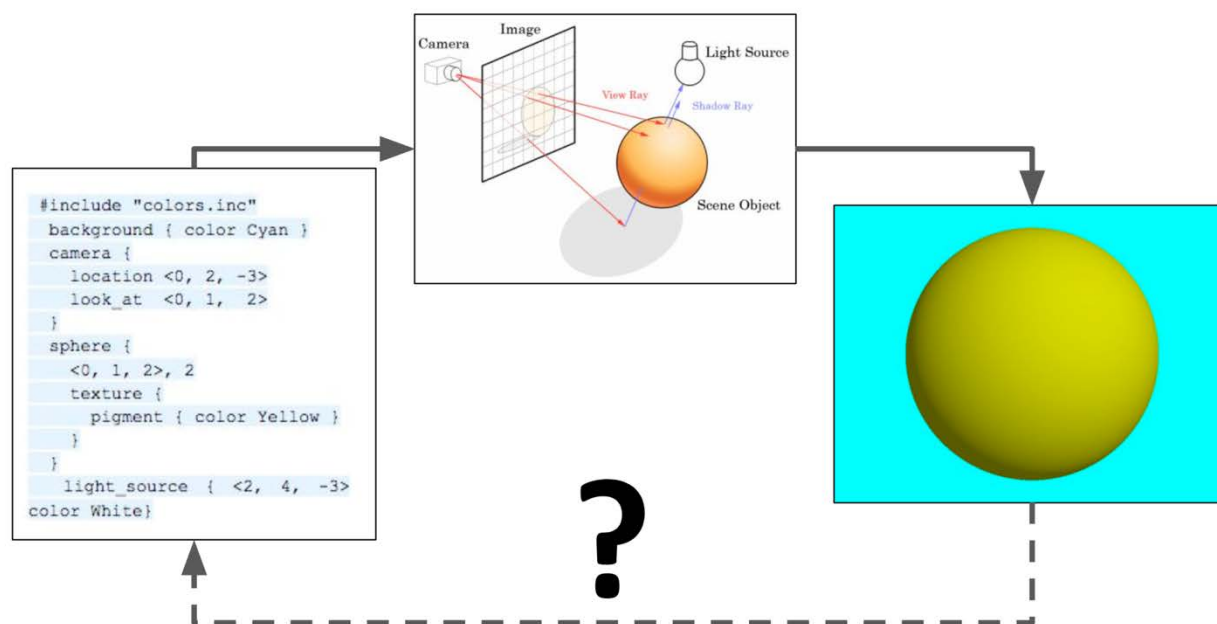
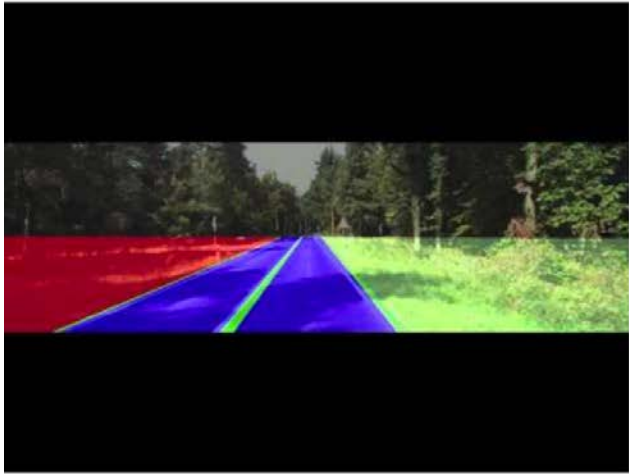


Figure 17. Computer vision (bottom path) as the inverse problem to computer graphics (top path).

Inverse graphics



Bottom-up computer vision



Method	Accuracy
Aly et al	68.31%
GPGP (Best Single Appearance)	64.56%
GPGP (Maximum Likelihood over Multiple Appearances)	74.60%

Figure 18. An illustration of generative probabilistic graphics for 3D road finding. On the left, results from Aly et al (2006). On the right, typical inference results from the proposed generative probabilistic graphics approach. (Kulkarni et al., 2015 [2])

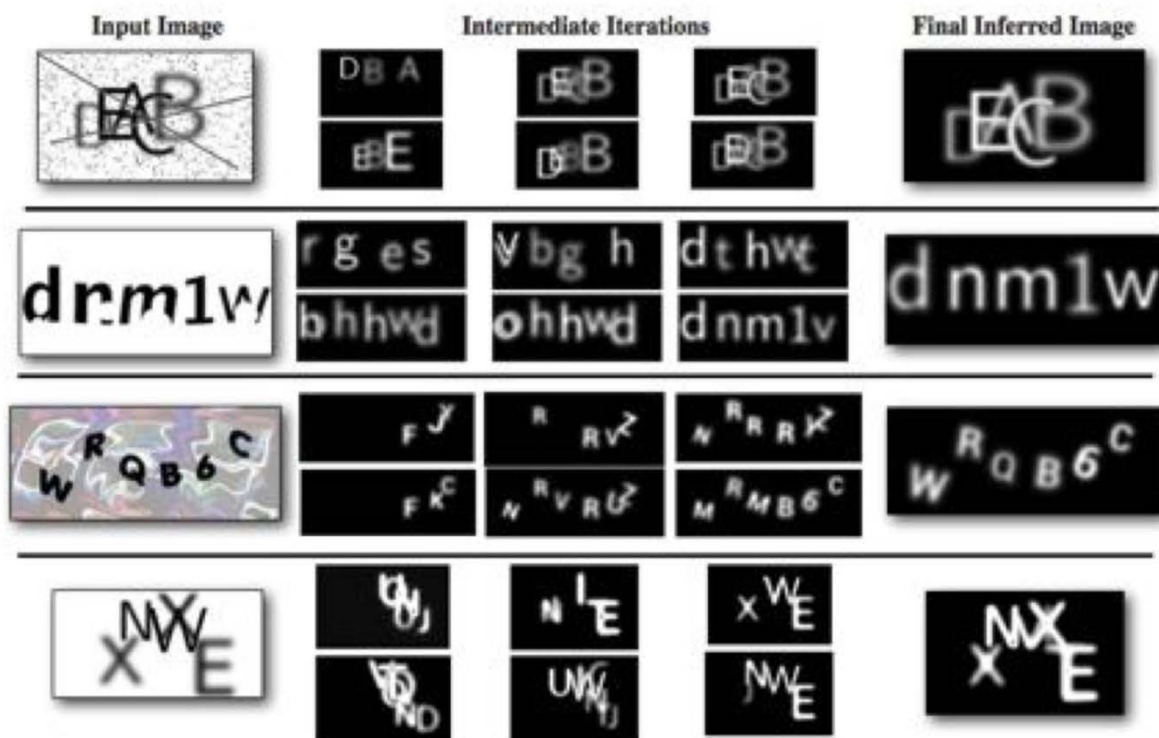


Figure 19. Four input images from our CAPTCHA corpus, along with the final results and convergence trajectory of typical inference runs. The first row is a highly cluttered synthetic CAPTCHA exhibiting extreme letter overlap. The second row is a CAPTCHA from TurboTax, the third row is a CAPTCHA from AOL, and the fourth row shows an example where our system makes errors on some runs. Our probabilistic graphics program did not originally support rotation, which was needed for the AOL CAPTCHAs; adding it required only 1 additional line of probabilistic code. (Mansinghka et al., 2013)

2.5 MetaProb

Probabilistic programming is based on two ideas: (i) representing probabilistic models as programs and (ii) representing inference and learning operations as meta-programs. Most existing probabilistic languages provide a small set of built-in meta-programs; adding new inference and learning techniques requires extending the language implementation. Metaprob is a probabilistic meta-programming language that aims to remove this limitation. In Metaprob, partial execution traces of probabilistic programs are first-class objects. The Metaprob interpreter can execute probabilistic programs against partial traces, treating traces as a source of Pearl-style interventions on the behavior of a probabilistic program. Metaprob (meta-) programs can also trace the behavior of other probabilistic programs, modify these traces, and use interventions based on these modified traces to implement Monte Carlo inference strategies. This research has shown how to use Metaprob's language constructs to implement two distinctive features of the Church probabilistic programming language—namely, stochastic memorization and Metropolis-Hastings transitions on execution traces—as ordinary user-space meta-programs.

From a practical standpoint, Metaprob is an unusually suitable vehicle research and teaching of probabilistic programming and meta-programming, much like Lisp was a suitable vehicle for research and teaching in both programming techniques and in language design, interpretation, and compilation. Metaprob also represents our best attempt to solve a fundamental problem posed by Dr. Kathleen Fisher in the early stages of the PPAML program: is there an intermediate representation that could potentially be shared among multiple probabilistic programming languages, and a simple interface that allows different ``solvers" for inference problems to interact and cooperate?

This report gives key examples of Metaprob programs that highlight the principles of probabilistic programming that we discovered and developed over the course of the PPAML program.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

For all the research we performed in this program, we adopted an experimental approach driven by the evaluation scenarios. Our overarching goal was to formalize and simplify the development of systems that perform probabilistic modeling and inference. The metrics used to evaluate our progress include the lines of code required to specify problems and the runtime and accuracy of the resulting solutions.

For our research papers and program presentations, we chose to evaluate our developed systems on both internally developed benchmark problems, on benchmark problems chosen by and on DARPA challenge problems developed by the TA1 Galois team. Our internal experiments and our interactions with DARPA evaluations led us to pursue an approach involving multiple probabilistic language implementations --- Venture, BayesDB, Picture, and Metaprob --- rather than a single monolithic system.

Regular engineering meetings and research collaboration meetings were carried out between the teams working together on this program. Additional feedback was sought by meetings with academic and corporate partners, presentations at academic conferences, and academic courses.

During the course of the program, we devoted a major effort to the packaging, testing, and usability of the system. Our releases included both unit and system tests (comprising our regression test). We also devoted major effort to documentation included with system deliverables. Finally, a major effort was devoted to open-sourcing and releasing to the public our analyses, models, and tools.

All of the probabilistic programming languages that we developed run on open-source software and standard hardware; no proprietary software or hardware is required.

4.0 RESULTS AND DISCUSSION

This section presents results from a broad class of probabilistic programs in multiple languages that were developed over the course of the program. It includes results on program challenge problems developed by DARPA, as well as results on problems that were developed internal to our team for testing and benchmarking purposes.

1. **Results for solving inference problems using custom inference strategies written in Venture.** These include quantitative results for a set of benchmark problems, some of which were inspired by the PPAML "Small Problems" benchmark set developed by Galois as part of the challenge problems.
2. **Example probabilistic programs and meta-programs written in Metaprob.** Metaprob is a probabilistic meta-programming language that was initially developed as a set of extensions to Venture.
3. **Example results drawn from program challenge problems,** specifically (i) the January 2017 Hackathon on the Gapminder database and (ii) Challenge Problem 7: Flu forecasting.
4. **Example results for our team challenge problem of analyzing a database of Earth satellites using BayesDB.** This section also includes background material on BayesDB, including a brief overview of its architecture and pointers to key technical innovations. It also illustrates example capabilities that are analogous to those evaluated by DARPA during the PPAML Hackathons.
5. **Example results showing how probabilistic programming can be used to reimplement the "Automatic Statistician" in under 70 lines of code in Venture .** These results show that probabilistic meta-programs written in Venture can solve hard model discovery problems.
6. **Example results showing how probabilistic programming can be used to solve problems of visual scene understanding.** These emphasize results from Picture, a domain-specific probabilistic programming language we developed for this type of problem, motivated in part by DARPA's interest in the domain.

4.1 Results for solving inference problems using custom inference strategies written in Venture².

Today, practitioners typically solve probabilistic modeling and inference problems by developing custom inference strategies adapted to each problem. This is true in a broad class of

² This section is largely taken from Mansinghka et al. (2018, in review)

fields (Gelman et al., 2014; Hahnel et al., 2003; Liu, 2008; Murphy, 2012; Russell and Norvig, 2003; Thrum, Burgard, and Fox, 2005).

Many inference strategies work by breaking inference problems down into subproblems and repeatedly applying appropriate probabilistic inference tactics (algorithms) to each subproblem. The tactics and subproblems are chosen so that the results converge to a good answer to the overall inference problem. These strategies are now a standard approach and have been shown to deliver significant accuracy and performance benefits in this context (Andrieu et al., 2003; Huang, Tristan, and Morrisett, 2017; Lindsten, Jordan, and Schoen, 2014; Murray, 2013; Neal, 2003; Ranganath, Gerrish, and Blei, 2014; Wingate and Weber, 2013).

Current probabilistic programming languages, however, typically provide only a specific black-box inference strategy or set of strategies (Carpenter et al., 2016; Goodman et al., 2008; Goodman & Stuhlmueeller, 2014; Gordon et al., 2014; Gordon et al., 2014b; Uber AI Labs, 2017; Milch et al., 2007; Tolpin, van de Meent, and Wood, 2015; Tristat et al., 2014). This situation denies developers the substantial performance and accuracy benefits that custom inference strategies can provide. To the best of our knowledge, the only exception is the Venture language (Mansinghka, Selsam, and Perov, 2014).

Custom Inference Strategies: In this segment of the report we present new probabilistic programming language constructs that enable developers to specify custom inference strategies. These constructs allow developers to solve probabilistic inference problems by identifying subproblems and applying specified probabilistic inference tactics to these subproblems. Subproblems are specified by identifying a subset of target stochastic choices made by a probabilistic program and subset of absorbing stochastic choices that insulate the remaining computation from the effects of any changes in the target stochastic choices. Together, these target choices and absorbing choices define the boundaries of the subproblem. Inference tactics work within these boundaries, leaving the computation outside the boundaries unaffected. Once the subproblem is identified, the developer can then specify an inference tactic to use on that subproblem. Together, these constructs provide a flexible and expressive mechanism for specifying custom inference strategies.

We have implemented these constructs in the context of the Venture probabilistic programming language and used them to compare the performance of multiple inference strategies for solving a set of benchmark problems. Our experimental results indicate that no single inference strategy is optimal for all problems. These results highlight how custom inference strategies can improve inference programming and accuracy in probabilistic programming languages.

We also formalize the use of subproblem selection and inference tactics for probabilistic programs. This formalism provides a framework for integrating difference inference algorithms/solvers into a unified probabilistic programming environment. The solvers then become composable building blocks that developers can use to implement custom inference strategies that solve the probabilistic inference problem at hand.

This research yielded two contributions:

1. *Custom inference strategies.* It showcases probabilistic programming constructs that enable developers working in probabilistic programming languages to specify custom inference strategies. These custom strategies enable developers to identify subproblems and apply specified inference tactics to each subproblem. These constructs make it possible, for the first time, for developers using probabilistic programming languages to specify custom inference strategies for solving probabilistic modeling and inference problems.
2. *Experimental Results.* We have implemented a set of benchmark probabilistic computations in Venture. Experimental results from these benchmark computations illustrated the performance and accuracy benefits that custom inference strategies can provide in this context.

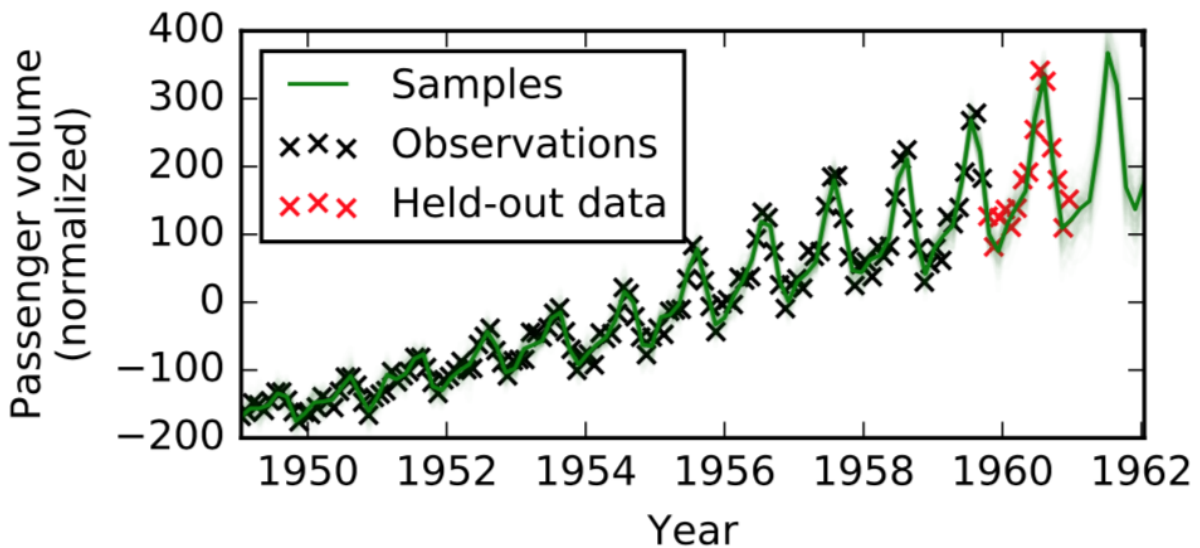


Figure 20. Gaussian Process Structure Learning using an inference strategy that applies Metropolis-Hastings to subproblems. Note the accuracy of the forecast (green) outside of the regime in which data was observed (black xs).

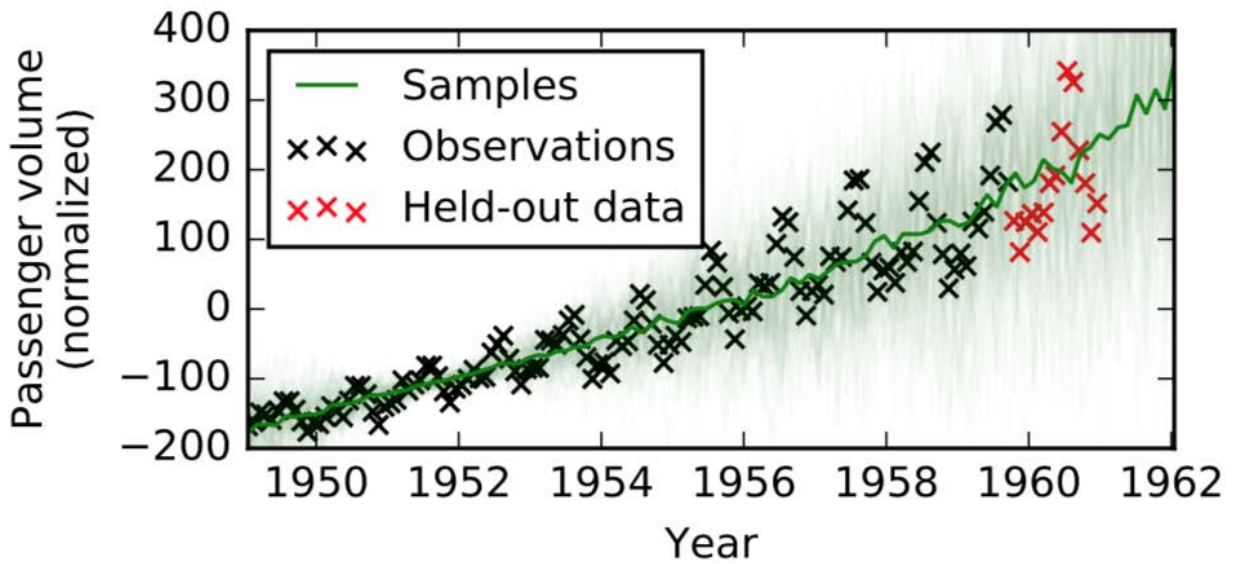


Figure 21. Gaussian Process Structure Learning with a different inference strategy. This inference result shows the need for custom inference strategies. The global Metropolis-Hastings strategy shown here captures the linear trend of the data but does not find the finer-grained structure in the time series.

```

assume tree_root = () -> {1};
assume get_hyper_prior ~ mem((node_index) -> {
  -log_logistic(log_odds_uniform() #hypers:node_index)
});
assume choose_primitive = (node) -> {
  base_kernel ~ categorical(simplex(.2, .2, .2, .2, .2),
    ["WN", "C", "LIN", "SE", "PER"])
  #structure:pair("base_kernel", node);
  cond(
    (base_kernel == "WN")
      ([["WN", get_hyper_prior(pair("WN", node))]]),
    (base_kernel == "C")
      ([["C", get_hyper_prior(pair("C", node))]]),
    (base_kernel == "LIN")
      ([["LIN", get_hyper_prior(pair("LIN", node))]]),
    (base_kernel == "SE")
      ([["SE", .01 + get_hyper_prior(pair("SE", node))]]),
    (base_kernel == "PER")
      ([["PER",
        .01 + get_hyper_prior(pair("PER_l", node)),
        .01 + get_hyper_prior(pair("PER_t", node))
      ]])
  ));
assume choose_operator = mem((node) -> {
  operator_symbol ~ categorical(simplex(0.45, 0.45, 0.1),
    ["+", "*", "CP"]) #structure:pair("operator", node);
  if (operator_symbol == "CP") {
    [operator_symbol, get_hyper_prior(pair("CP", node))]
  } else {
    operator_symbol
  }
});
assume generate_random_program = mem((node) -> {
  if (flip(.3) #structure:pair("branch", node)) {
    operator ~ choose_operator(node);
    [
      operator,
      generate_random_program(2 * node),
      generate_random_program(2 * node + 1)
    ]
  } else {
    choose_primitive(node)
  }
});

```

(a) Synthesis model: AST prior

```

assume produce_covariance = (source) -> {
  cond(
    (source[0] == "WN") (gp_cov_scale(source[1], gp_cov_bump)),
    (source[0] == "C") (gp_cov_const(source[1])),
    (source[0] == "LIN") (gp_cov_linear(source[1])),
    (source[0] == "SE") (gp_cov_se(source[1]**2)),
    (source[0] == "PER")
      (gp_cov_periodic(source[1]**2, source[2])),
    (source[0] == "+") (
      gp_cov_sum(
        produce_covariance(source[1]),
        produce_covariance(source[2]))),
    (source[0] == "*") (
      gp_cov_product(
        produce_covariance(source[1]),
        produce_covariance(source[2]))),
    (source[0][0] == "CP") (
      gp_cov_cp(
        source[0][1], .1,
        produce_covariance(source[1]),
        produce_covariance(source[2]))));
};
assume produce_executable = (source) -> {
  baseline_noise = gp_cov_scale(.1, gp_cov_bump);
  covariance_kernel = gp_cov_sum(
    produce_covariance(source),
    baseline_noise);
  make_gp(gp_mean_const(0.), covariance_kernel)
};

```

(b) Synthesis model: AST interpreter

Figure 22. Venture model code for the Gaussian process structure learning program. This probabilistic model code defines a probabilistic model over structured Gaussian process models for time series data.

```

assume source ~ generate_random_program(tree_root());
assume gp_executable = produce_executable(source);
define xs = get_data_xs("./data.csv");
define ys = get_data_ys("./data.csv");
observe gp_executable(${xs}) = ys;

```

(a) Data observation program

```

for_each(arange(T), (_) -> {
  infer
    resimulation_mh(
      minimal_subproblem(random_singleton(/?hypers)));
  infer
    resimulation_mh(
      minimal_subproblem(random_singleton(/?structure)))
});

```

(b) Synthesis inference strategy: Subproblem-based, single-site MH

```

for_each(arange(T), (_) -> {
  infer resimulation_mh(minimal_subproblem(/*));
});

```

(c) Synthesis inference strategy: Subproblem-based, single-site MH

Figure 23. Venture observation and inference code for the Gaussian process structure learning program. This code shows different custom inference strategies for discovering Gaussian process model structures.

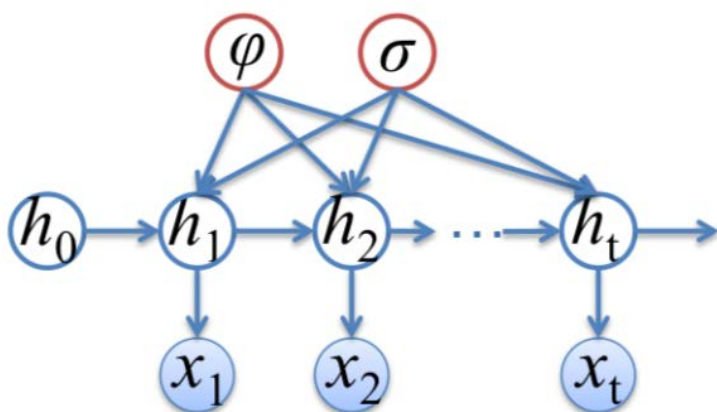


Figure 24. A graphical model representing the variables and dependencies for our second benchmark problem: a stochastic volatility model.

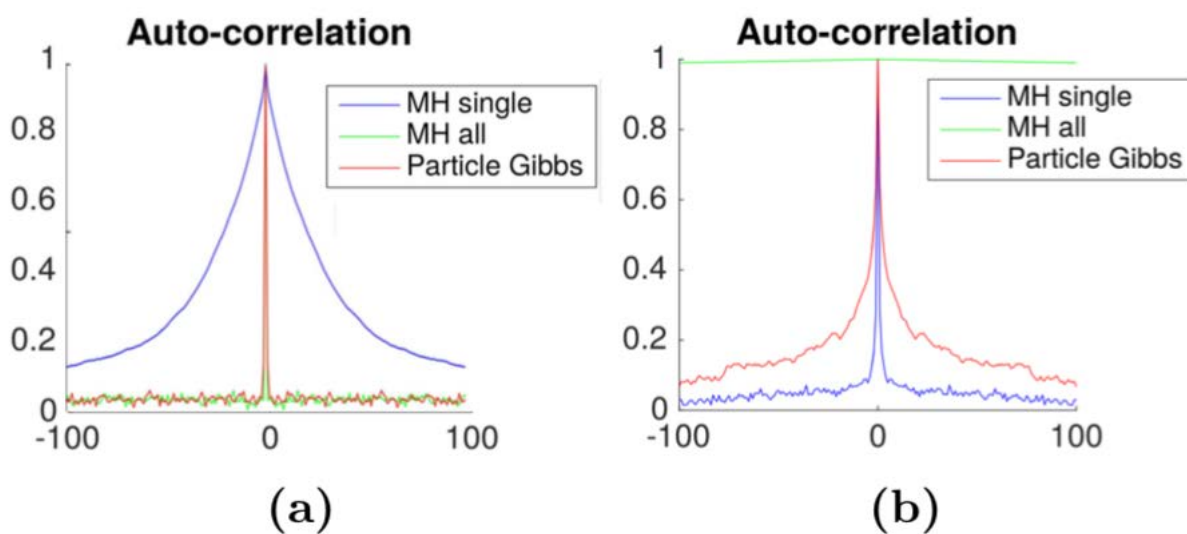


Figure 25. Stochastic Volatility Model with two types of observation sequences and three inference operators: single-site Metropolis-Hasting operating on individual random choices, global Metropolis-Hastings for all random choices, and particle Gibbs.


```

assume precision = 1 / 0.05**2;
assume sigma     = 1/sqrt(precision);
assume phi       = 0.99;
// sigma0 is fixed.
assume h = mem((t) -> {
  if(t <= 0) {
    normal(0 , sigma0) #h:t
  } else {
    normal(phi * h(t-1) * sigma , sigma) #h:t
  }
})
assume x = (t) -> {normal(0, h(t)/2)};
// Observe data.
for_each(arange(T),
  (t) -> {
    observe x(t) = X[t] //X[t] is the observation at time t
  })
// state estimation via Particle Gibbs with P particles for N
// iterations
for_each(arange(N),
  (_) -> {
    infer pgibbs(minimal_subproblem(ordered(/?h)), P)
  });
// state estimation via MH applied to single states every
// iteration for N * T * P iterations
for_each(arange(N * T * P),
  (_) -> {
    infer
      resimulation_mh(minimal_subproblem(random_singleton(/?h)))
  });
// state estimation via MH applied to all states every iteration
// for N * P iterations
for_each(arange(N * T * P),
  (_) -> {
    infer resimulation_mh(minimal_subproblem(/?h))
  });

```

Figure 26. Shows Venture code for the stochastic volatility model, along with code for custom inference.

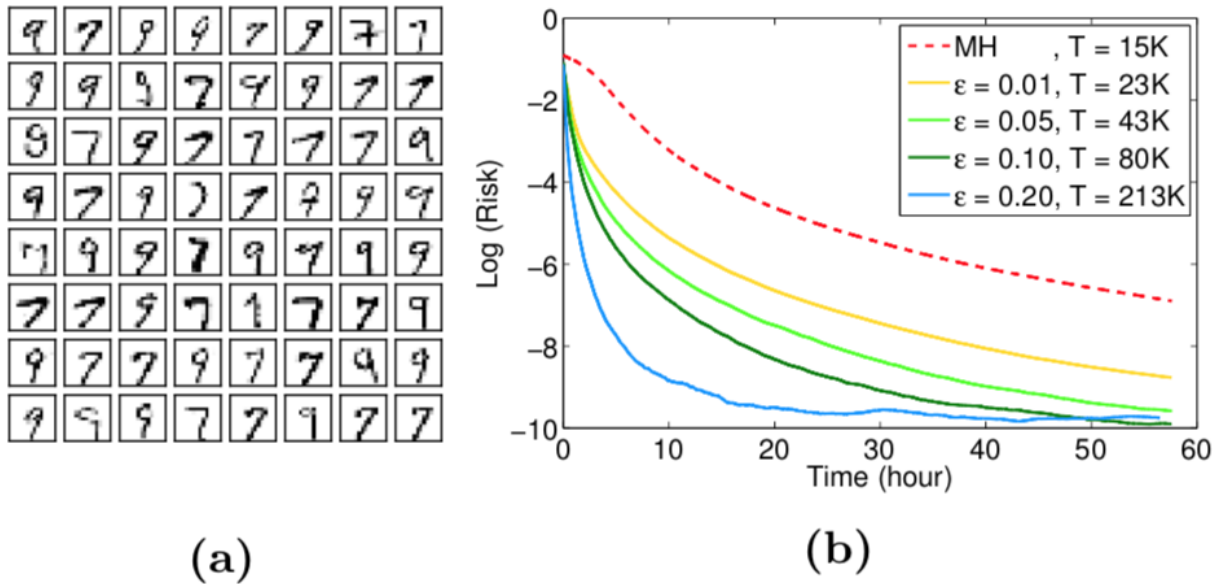


Figure 27. A third benchmark problem: logistic regression for classifying handwritten digits from the MNIST database.

```

assume w ~ multivariate_normal(mu, Sig) #w;
assume y_x = (x) -> {bernoulli(linear_logistic(w, x))};
// Observe data.
for_each(arange(size(data)),
  (n) -> {
    // Features X[n] = [x0_n, x1_n, ...], class Y[n]
    observe y_x(X[n]) = Y[n]
  })
// Do T iterations of MH with a prior proposal;
for_each(arange(T),
  (_) -> {
    infer resimulation_mh(minimal_subproblem(/*))
  });

// or do T iterations of MH with a random walk proposal of
// bandwidth sigma.
for_each(arange(T),
  (_) -> {
    infer
      mh_with_gaussian_drift_proposal(minimal_subproblem(/?w))
  });

```

Figure 28. Venture code for the Bayesian logistic regression example from the previous figure

```

// Fixed: dimensionality D, hypers mu_w, sig_w, m0, k0, v0, S0.
assume alpha ~ gamma(1, 1) #alpha;
assume crp    = make_crp(alpha);
assume z      = mem((i) -> {crp() #z:i});
assume w      = mem((z) -> {
    multivariate_normal(mu_w, sig_w) #w:z
});
assume c      = mem((z)) -> {
    make_collapsed_multivariate_normal(m0, k0, v0, S0)
};
assume x = (i) -> {c(z(i))};
assume y = (i, x) -> {bernoulli(linear_logistic(w(z(i))), x)};
// Observe data.
for_each(arange(size(data)),
    (n) -> {
        observe x(n) = X[n]; // X[n] is n-th feature vector.
        observe y(n) = Y[n]  // Y[n] is n-th class label.
    });
// T steps of MH for hyperparams, single-site (collapsed) gibbs
// for z, MH over weights for a randomly chosen cluster
define mh_with_gibbs = () -> {
    resimulation_mh(minimal_subproblem(/?alpha));
    gibbs(minimal_subproblem(random_singleton(/?z)));
    mh_with_gaussian_drift_proposal(minimal_subproblem(/?w))
};
for_each(arange(T),
    (_) -> {infer mh_with_gibbs()});
// or use the default inference operator for about the same
// amount of computations (two input clusters are inferred
// in the example).
for_each(arange(T * (1 + N + 2)),
    (_) -> {
        infer
            resimulation_mh(minimal_subproblem(random_singleton(/*)))
    });

```

Figure 29. Venture code for a more complex Dirichlet process mixture of experts model, extending the previous logistic regression benchmark.

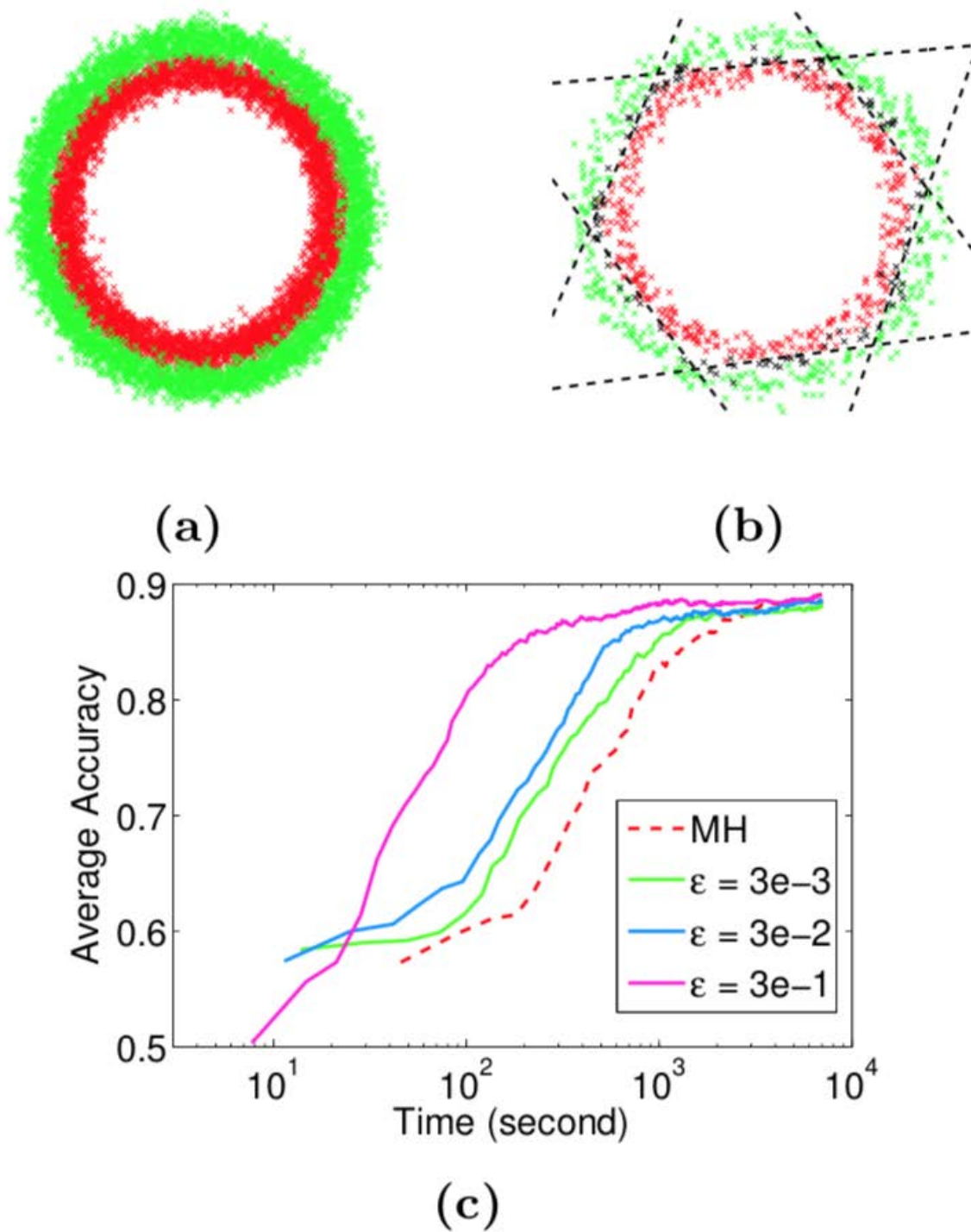


Figure 30. Accuracy results for the Dirichlet process mixture of experts. (a) training data. (b) prediction on test data with $\epsilon = 0.3$ after 2 hours. 6 clusters are found with decision boundaries (dashed lines) and mis-classified points (black dots). (c) Predictions accuracy vs running time in log domain.

```

// Model component.
assume disease_prior = 0.01;
assume disease_1 ~ flip(disease_prior) #symptomA:0;
assume disease_2 ~ flip(disease_prior) #symptomA:1;
assume disease_3 ~ flip(disease_prior) #symptomA:2 #symptomB:0;
assume disease_4 ~ flip(disease_prior) #symptomB:1;
assume disease_5 ~ flip(disease_prior) #symptomB:2;
assume disease_6 ~ flip(disease_prior) #symptomB:3;
assume get_causal_link = (parent) -> {
  if (parent) {
    0.01 // Probability of a leak
  } else {
    1.
  }
};
assume get_effect = (parents, effect) -> {
  if (size(parents) == 0) {
    effect
  } else {
    get_effect(
      rest(parents),
      effect * get_causal_link(first(parents))
    )
  }
};
assume noisy_or = (parents) ~> {
  p_spontaneous = 0.001;
  flip(
    1 - ((1 - p_spontaneous) * get_effect(to_list(parents), 1))
  )
};

```

```

// Observation component.
observe
  noisy_or([disease_1, disease_2, disease_3]) = True;
observe
  noisy_or([disease_3, disease_4, disease_5, disease_6]) = True;
// Inference component: global MH.
for_each(arange(n_inference_steps),
  (_) -> {
    infer
      resimulation_mh(minimal_subproblem(/*))
  });
// Inference component: SSMH.
for_each(arange(n_inference_steps),
  (_) -> {
    infer
      resimulation_mh(minimal_subproblem(random_singleton(/*)))
  });
// Inference component: single-site Gibbs sampling.
for_each(arange(n_inference_steps),
  (_) -> {
    infer
      gibbs(minimal_subproblem(random_singleton(/*)))
  });
// Inference component: Gibbs sampling -- block proposals.
for_each(arange(n_inference_steps),
  (_) -> {
    infer
      gibbs(minimal_subproblem(/*symptomA));
      gibbs(minimal_subproblem(/*symptomB))
  });

```

Figure 31. Venture model component, observation component, and inference components for a bipartite Bayesian network inference problem.

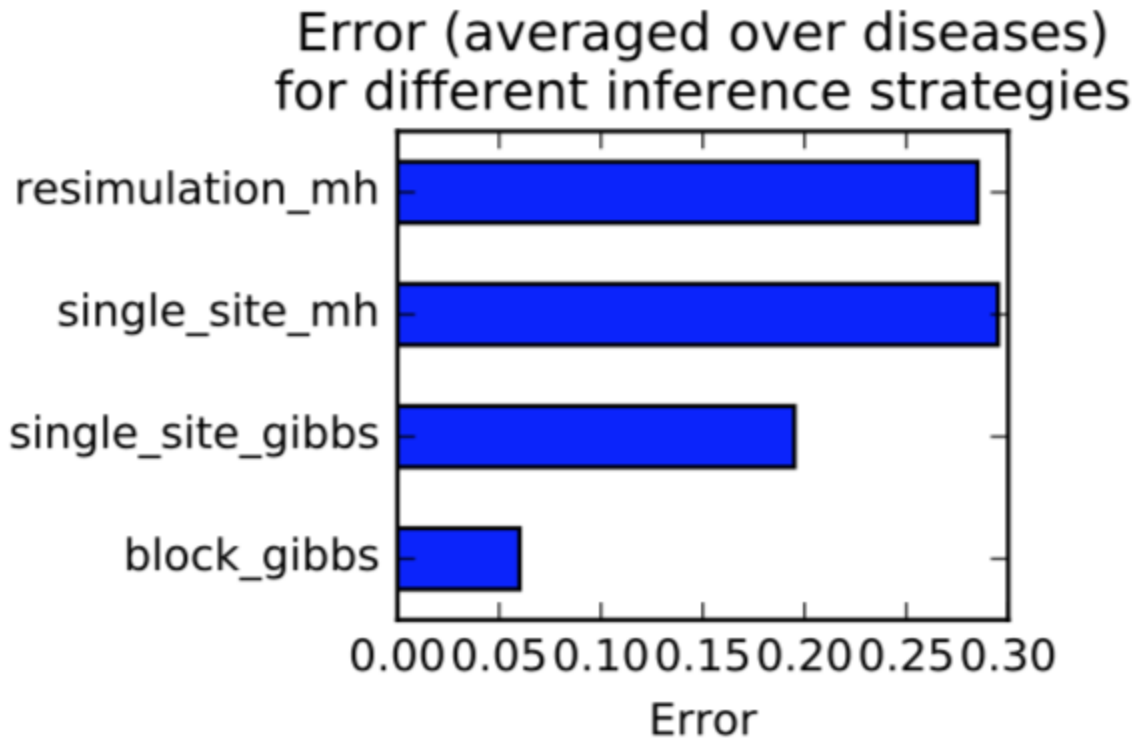


Figure 32. A comparison of the accuracy of custom inference strategies on the bipartite Bayesian network. Inference accuracy is calculated as the average difference in probabilities of each state obtained from a single Markov chain run for 100 samples and the ground truth probabilities.

Here we have presented language constructs for specifying custom probabilistic inference strategies in probabilistic programming languages. These constructs enable, for the first time, developers using probabilistic programming languages to identify subproblem and apply specified inference tactics to the identified subproblems. Results from benchmark programs implemented in the Venture probabilistic programming language highlight the significant accuracy and efficiency benefits that the use of customized inference strategies can deliver in this context.

4.2 Example probabilistic programs and meta-programs written in Metaprob.

This section gives example probabilistic programs and meta programs written in MetaProb; most of the following segment of the report is taken from Radul and Mansinghka (2018, expected).

Probabilistic programming is based on two ideas: (i) probabilistic models can be represented as executable computer programs and (ii) inference and learning algorithms can be represented as meta-programs that take these programs as inputs and/or produce them as outputs. These ideas are the basis of a rapidly growing collection of languages and implementations. Each language provides some “built-in” probability distributions, plus language constructs for combining them to larger programs that represent meaningful probabilistic models. Each language also provides a

small set of “built-in” inference algorithms that can infer likely outcomes from these probability distributions given data.

Unfortunately, existing languages do not provide language constructs for adding new probability distributions or inference algorithms. If the built-in set is inadequate for some problem, the probabilistic programmer needs to either choose a different language or learn how to extend the language implementation. Most existing languages also do not provide constructs for combining built-in general-purpose algorithms with user-specified algorithms that are customized for a given application. These are fundamental limitations. In practice, they restrict the use of probabilistic programming to applications where a small set of probability distributions and generic inference algorithms are known to be adequate ahead of time. They also greatly limit the extent to which probabilistic programmers can take advantage of advances in algorithms for approximate inference.

Metaprob is a simple, extensible language for probabilistic programming and meta-programming. Metaprob is simple enough that an implementation of the language fits on a single page, requiring under 100 lines of Metaprob code. Also, in Metaprob, new probability distributions and inference algorithms can be written as ordinary user-space programs. Metaprob is extensible and expressive enough that distinctive features of other probabilistic languages, such as Metropolis-Hastings inference, sequential Monte Carlo inference, and modeling with Dirichlet processes, can each be written in under one page of Metaprob code. Inference in Metaprob is not limited to Monte Carlo sampling.

It is helpful to view MetaProb in terms of an analogy to Lisp. Lisp is well known both for its unusual simplicity and for its support for meta-programming. Lisp source code follows a small set of rules, and is easy to represent using lists, the central data structure in the language. The process of Lisp program execution also follows a small set of rules that closely mirror the structure of Lisp source code. Thus Lisp interpreters can be written as short Lisp programs. Executable Lisp programs can also be represented in the language, as “procedures” that take a specified list of formal parameters. These procedures are ordinary data Lisp objects that can be produced and executed by other Lisp programs. This combination of simplicity and expressiveness helped make Lisp a widely used language for research in artificial intelligence and in programming language design and implementation. Lisp also provides a simple model of computation that has been useful for teaching.

The design of Metaprob aims to preserve the simplicity and expressiveness of Lisp, while adding the minimal additional features needed for research in probabilistic artificial intelligence and in probabilistic programming. Metaprob also aims to serve as a simple model of probabilistic computation that can be used to teach the key concepts to a broad audience. This model of computation differs from the Lisp model in three fundamental ways:

1. The built-in Metaprob interpreter allows the execution of a Metaprob program to be intervened on, i.e., forced to take on specific values at chosen points in a program’s execution. Metaprob provides language constructs for naming points in a program’s execution and a data structure called a trace for storing mappings between execution points and values. Metaprob’s syntax is designed to make it easy to use these capabilities.

2. Probabilistic programs in Metaprob constitute extensible packages of executable code, source code, meta-data, and meta-programs. These packages can be re-opened by other Metaprob (meta)programs, for example to retrieve the source code of a program or to retrieve a program that specifies the output probability distribution of some other program.

3. Typical Metaprob programs rely on a standard library¹ of “inference meta-programs”, not just a built-in interpreter or compiler. These inference meta-programs take a probabilistic program and also a “target” trace as input, and generate executions of the program that are compatible with the “target” trace.

These differences constitute the core contributions of Metaprob’s design. Here, we introduce the language features that underlie these differences and shows that they are sufficient to implement a broad class of techniques for probabilistic modeling and inference. Applications are drawn from hierarchical Bayesian modeling, Bayesian networks, and non-parametric Bayesian statistics. To the best of our knowledge, Metaprob is the first probabilistic language with an explicit meta-circular implementation, and the only probabilistic language in which the distinctive modeling and inference features of expressive languages such as Church can be implemented as short (meta)programs. We also believe that Metaprob is the only probabilistic language with support for interventions and for adding new probability distributions within the language.

The remainder of this section gives a tutorial introduction to Metaprob. It is also helpful to navigate this section in light of the main Metaprob example programs that it presents. The full set of example programs we have developed for Metaprob so far include the following:

1. Inferring if a coin is tricky or fair via a Monte Carlo inference in a hierarchical Bayesian model.
2. Comparing inference given observations to inference given interventions on a discrete-variable Bayesian network, using both approximate and exact inference algorithms.
3. Implementing an interpreter for Metaprob that supports overriding the execution of a program via interventions.
4. Implementing a tracing interpreter for Metaprob that records the stochastic choices made by a probabilistic program during its execution.
5. Adding the Gaussian probability distribution, packaging code for a sampler with code for the log output probability density of the Gaussian.
6. Performing inference via rejection sampling and sampling-importance-resampling, i.e. via sampling, weighting, and resampling traces based on their compatibility with a trace representing the constraints on the execution of a program.
7. Implementing an approximate inference meta-program that changes a single random choice at the time. This program re-implements the single-site Metropolis-Hastings

inference algorithm that is the basis of probabilistic programming languages such as Church and WebPPL.

8. Implementing an inference algorithm that enumerates all possible executions of probabilistic programs with finite support and deterministic control flow.
9. Adding the Chinese restaurant process, and adding a memoizer to the language, both of which use traces to store their internal state.
10. Putting these pieces together to build a Chinese restaurant process mixture model and using it to infer clusters from numerical data. This example was a capstone example in the original paper on the Church probabilistic programming language that helped to crystallize growth of the probabilistic programming field.

```

// (A) Define a function whose traces will serve as the model
flip_coins = (n) ~> {
  root_addr = &this;
  tricky = flip(0.1);
  weight = if (tricky) {uniform(0, 1);} else {0.5;};
  map((i) ~> with_address /$root_addr/datum/$i/: flip(weight),
      range(n));
};

// (B) Run it once
a_trace = {{ }};
trace_choices(
  program          = flip_coins,
  inputs           = [5],
  intervention_trace = {{ }},
  output_trace      = a_trace);

// (C) Have a look at (a sample of) whether the coin is tricky
print(*a_trace[/1/tricky/flip/]);
False

// (D) Force the coin to come up heads every time
a_trace[/datum/0/flip/] := true;
a_trace[/datum/1/flip/] := true;
a_trace[/datum/2/flip/] := true;
a_trace[/datum/3/flip/] := true;
a_trace[/datum/4/flip/] := true;

// (E) Define a Metropolis-based inference strategy
approximate_inference_update = () ~> {
  single_site_metropolis_hastings_step(
    program = flip_coins,
    inputs  = [5],
    trace   = a_trace,
    constraint_addresses = set_difference(
      addresses_of(a_trace),
      [/1/tricky/flip/, /2/weight/then/0/uniform/]));
};

// (F) Run it a bit
repeat(times = 20, program = approximate_inference_update);

// (G) Look at (a sample of) the (inferred) trickiness
print(*a_trace[/1/tricky/flip/]);
False

```

Figure 33. A Metaprob transcript, modeling n flips of a potentially biased coin.

In this example, the prior is that the coin has an 0.1 probability of being biased, in which case it has an unknown uniformly distributed weight; otherwise is fair. Each output is a sample from the distribution on values of the tricky variable at that point; they will vary with the initial entropy

given to the pseudo-random number generator. (Radul and Mansinghka, 2018).

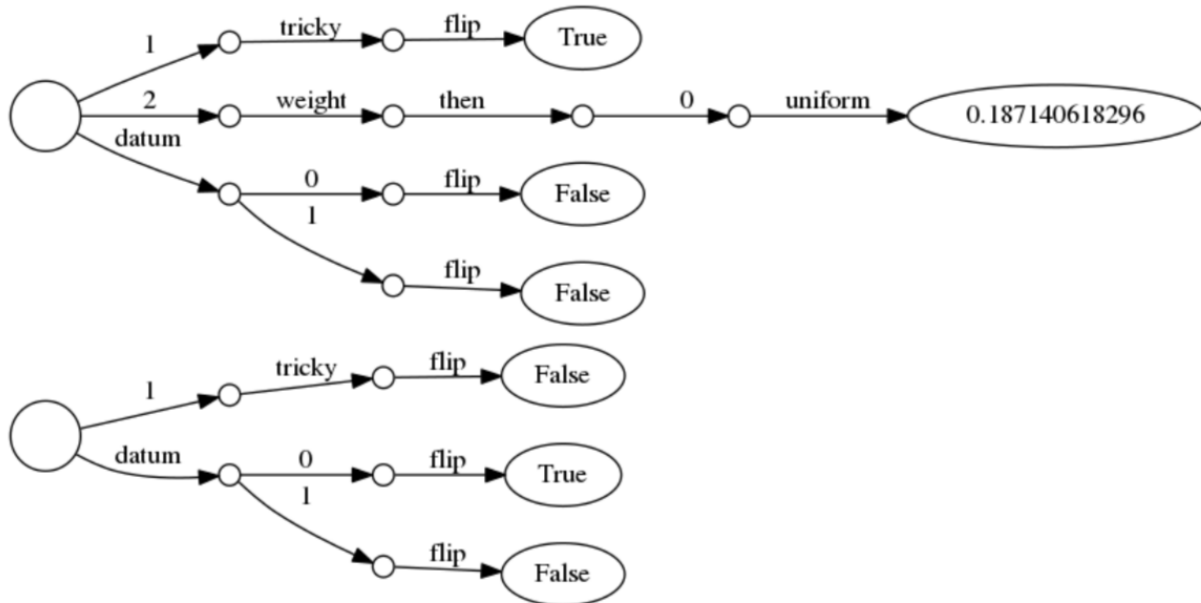


Figure 34. Two traces of executing the flip_coins program from Figure 1 for two flips.

The example in the top pane takes the “tricky coin” control flow path; note the coin weight drawn from the Beta distribution. The example in the bottom pane takes the “fair coin” control flow path. The coin weight (0.5) is not explicitly recorded there because it is deterministic in the source code. The large node on the left is the root of the trace. We can name any node (or subtrace) by listing the edge labels that lead to it from the root. This is an address, which we write slash-bounded, by analogy with URLs. For example, the Boolean indicating whether the coin is tricky lives at the address /1/tricky/flip/ (Radul and Mansinghka, 2018).

```

earthquake_bayesian_network = () -> {
  earthquake = flip(0.1);
  burglary   = flip(0.1);
  p_alarm =
    if (burglary && earthquake) 0.9
    else if (burglary) 0.85
    else if (earthquake) 0.2
    else 0.05;
  alarm      = flip(p_alarm);
  p_john_call = if (alarm) 0.8 else 0.1;
  john_call  = flip(p_john_call);
  p_mary_call = if (alarm) 0.9 else 0.4;
  mary_call  = flip(p_mary_call);
  "ok";
};

query = (state) -> {
  earthquake = *state[/0/earthquake/flip/];
  burglary   = *state[/1/burglary/flip/];
  alarm      = *state[/3/alarm/flip/];
  john_call  = *state[/5/john_call/flip/];
  mary_call  = *state[/7/mary_call/flip/];
  (earthquake, burglary, alarm, john_call, mary_call);
};

```

Figure 35. Definition of the classic earthquake-burglary Bayes net in Metaprob, including a function for collecting the variables of interest from a trace of its execution. (Radul and Mansinghka, 2018)

```

alarm_went_off = {{ }};
alarm_went_off[/3/alarm/flip/] := true;

inter_trace = () ~> {
  t = {{ }};
  trace_choices(
    program      = earthquake_bayesian_network,
    inputs       = [],
    intervention_trace = alarm_went_off,
    output_trace  = t);
  t;
};

inter_traces = replicate(
  times = n_samples, program = inter_trace);

discrete_histogram(
  traces      = inter_traces,
  label_func  = query,
  title       = "Alarm_intervened_(samples)");

```

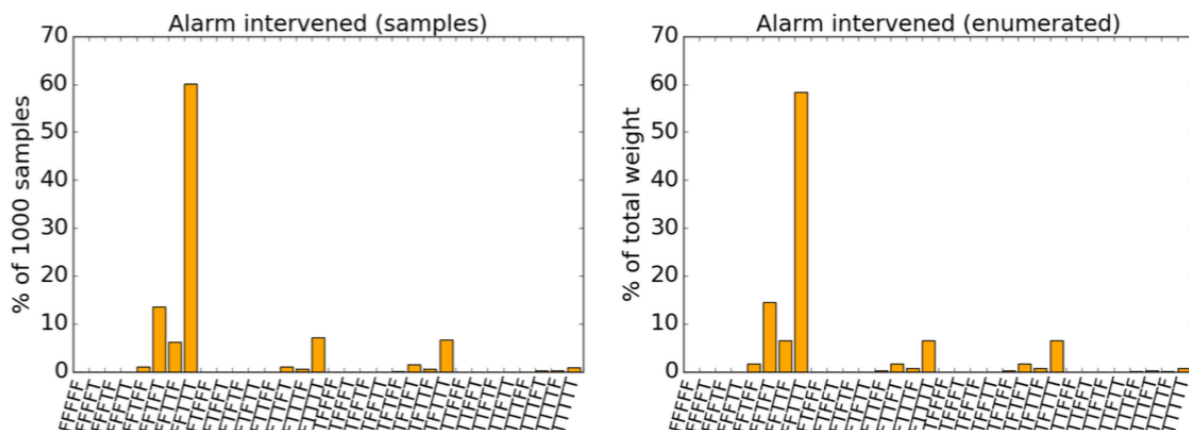
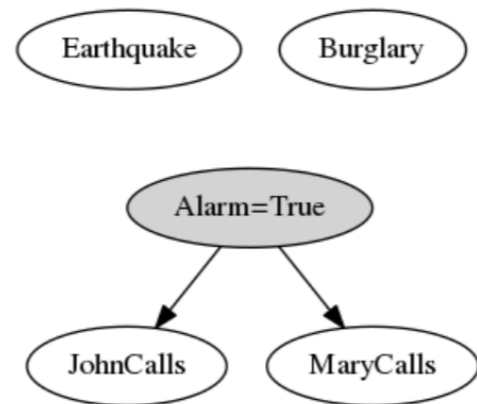


Figure 36. The joint distribution induced by intervening on the earthquake-burglary Bayes net to set the alarm variable to true.

The top row shows code for sampling from the distribution on traces of this Bayes net (top left) and a graphical representation of the network induced by the intervention (top right). The bottom row shows, from left to right, the empirical distribution of sampling from it and the exact distribution computed by enumerating over possible states.




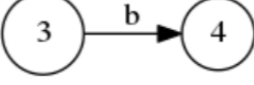

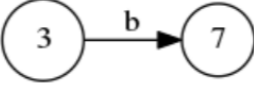
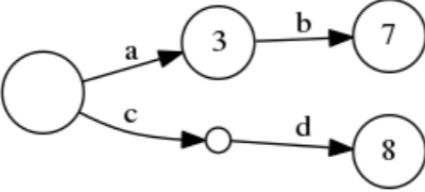
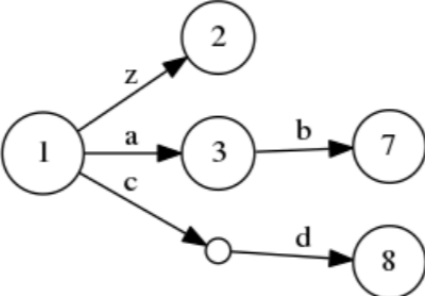
Metaprob syntax	Semantic effect	Result
<code>trace = {{ }};</code>	Literal empty trace	
<code>trace[/a/] := 3;</code>	Assign 3 at the key a	
<code>trace[/a/b/] := 4;</code>	Assign 4 at the two-key address /a/b/	
<code>subtrace = trace[/a/];</code>	Obtain the subtrace at the key a	
<code>*subtrace;</code>	Fetch the value at the root node (of subtrace)	3
<code>trace[/a/b/] := 7;</code>	Repeated assignments overwrite	
<code>subtrace;</code>	The subtrace is an alias, so sees the modification	
<code>trace[/c/d/] := 8;</code>	Traces have unlimited fanout, and nodes may lack values	
<code>trace has_value;</code>	Postfix operator checking for a value at the root node	False
<code>addresses_of(trace);</code>	List of addresses that have values	/a/, /a/b/, /c/d/
<code>{{ 1, z = 2, a = **subtrace, */c/d/ = 8}};</code>	Trace literal syntax can include a root value, assignment by key, and splicing of subtraces (with **) or addresses (with *)	

Figure 37. Basic operations on Metaprob traces, showing the Metaprob cod and results.

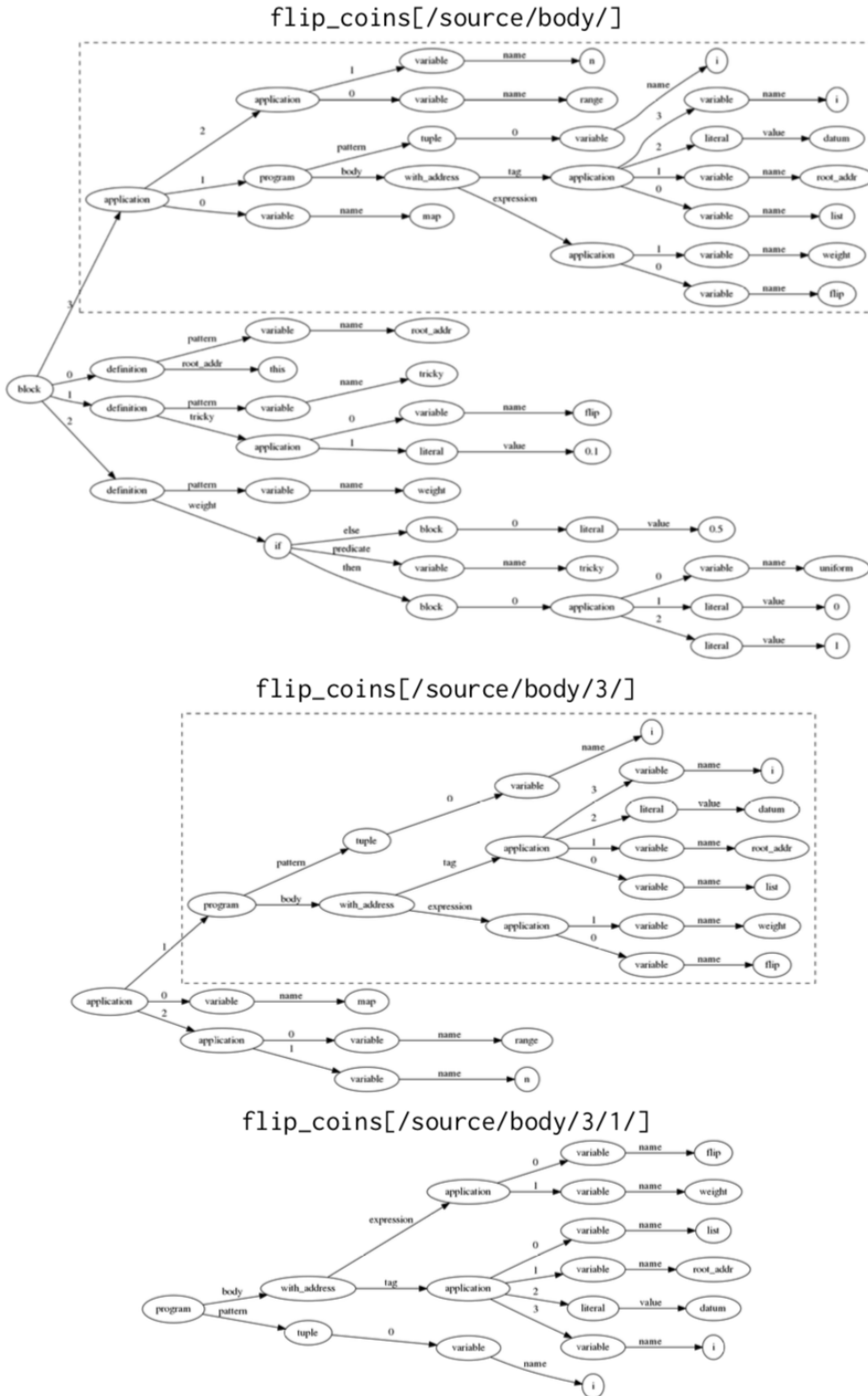


Figure 38. An illustration of trace prefix selection. Each dashed box highlights which subtrace is shown in the subsequent pane.

```

single_site_metropolis_hastings_step =
(program, inputs, trace, constraint_addresses) ~>
{
  // Choose a target site to propose uniformly
  choice_addresses = addresses_of(trace);
  candidates       = set_difference(choice_addresses,
                                   constraint_addresses);
  target_address   = uniform_sample(candidates);

  // Compute a proposal trace
  initial_value    = *trace[target_address];
  initial_num_choices = length(candidates);
  del trace[target_address];
  new_trace = {{ }};
  (_, forward_score) = propose_and_trace_choices(
    program      = program,
    inputs       = inputs,
    intervention_trace = {{ }},
    target_trace  = trace,
    output_trace  = new_trace);
  new_value      = *new_trace[target_address];

  // The proposal is to move from trace to new_trace;
  // Now compute the acceptance ratio.
  new_choice_addresses = addresses_of(new_trace);
  new_candidates       = set_difference(new_choice_addresses,
                                       constraint_addresses);
  new_num_choices      = length(new_candidates);
  restoring_trace = { { *target_address = initial_value } };
  for_each( set_difference(choice_addresses,
                          new_choice_addresses),
    (addr) -> { restoring_trace[addr] := *trace[addr]; } );
  del new_trace[target_address];
  (_, reverse_score) = propose(
    program      = program,
    inputs       = inputs,
    intervention_trace = restoring_trace,
    target_trace  = new_trace);
  new_trace[target_address] := new_value;
  log_p_accept = forward_score - reverse_score
    - log(initial_num_choices) + log(new_num_choices);

  if (log(uniform(0, 1)) < log_p_accept) { // Accept
    for_each( set_difference(choice_addresses,
                          new_choice_addresses),
      (addr) -> { del trace[addr]; } );
    for_each( new_choice_addresses,
      (addr) -> { trace[addr] := *new_trace[addr]; } );
  } else { // Reject
    trace[target_address] := initial_value;
  };
};

```

Figure 39. A user-space meta-program for performing a single step of “lightweight” Metropolis-Hastings inference. The program uses standard meta-programs for proposing program traces subject to constraints to create a proposal trace and calculate the acceptance ratio. (Radul and Mansinghka, 2018)


```

mcmc_sample = () ~> {
  result = {{ }};
  constraints = addresses_of(target);
  trace_choices(
    program          = normal_normal,
    inputs           = args,
    intervention_trace = target,
    output_trace      = result);
  repeat(times = s, program = () ~> {
    single_site_metropolis_hastings_step(
      program = normal_normal,
      inputs  = args,
      trace   = result,
      constraint_addresses = constraints);
  });
  *result[/0/x/gaussian/];
};

binned_histogram(
  samples = replicate(times = 20, program = mcmc_sample),
  overlay_densities =
    {{ prior = prior_density, posterior = analytic_density }},
  title = "MCMC_" + string(s) + "_resimulation_steps");

```

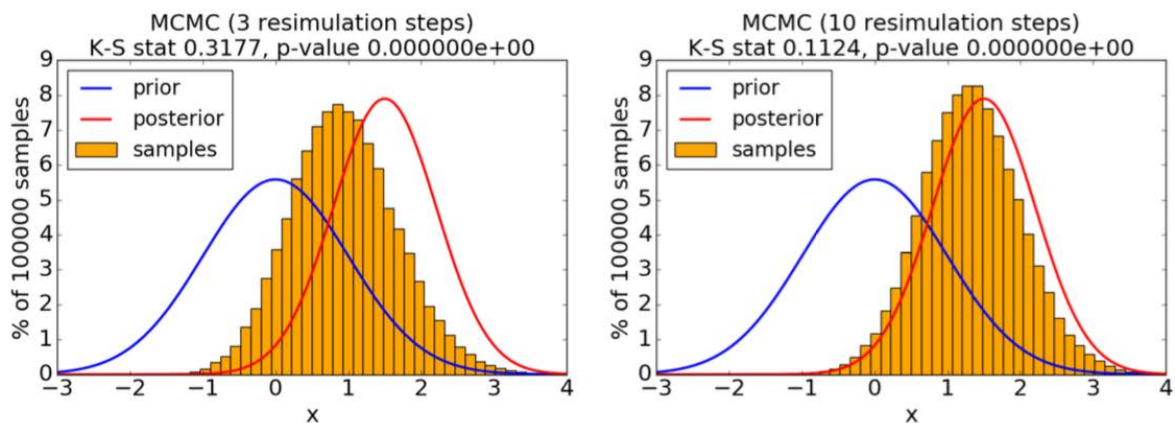


Figure 40. Markov-chain Monte Carlo inference over the user-space Gaussian distribution. The top panel is a transcript of invoking MCMC (with a prior resimulation proposal), and the bottom panel shows histograms of the samples thus obtained. (Radul and Mansinghka, 2018)

```

generate_from_dirichlet_process_mixture =
(num_datapoints) ~> {
  root_addr      = &this;

  alpha          ~ gamma(1.0, 1.0);
  sample_assignment = make_chinese_restaurant_sampler(alpha);
  get_cluster     = mem( (i) ~> { sample_assignment(); } );

  a              ~ inverse_gamma(3.0, 10.0);
  b              ~ uniform(0.0, 10.0);
  mu             ~ uniform(-150.0, 150.0);
  V              ~ inverse_gamma(2, 100);

  get_mu         = mem( (cluster) ~> {
    normal(mu, V × get_sigma(cluster));
  } );
  get_sigma      = mem( (cluster) ~> {
    sqrt(inverse_gamma(a, b));
  } );

  get_datapoint  = mem( (i) ~> {
    cluster = get_cluster(i);

    with_address /$root_addr/datum/$i/:
    normal( get_mu(cluster),
            get_sigma(cluster) );
  } );
  map(get_datapoint, range(num_datapoints));
};

```

Figure 41. Metaprob source code for a program that generates data from a Dirichlet process mixture of Gaussians with randomly chosen hyper-parameters. This program uses user-space implementations of `make_chinese_restaurant_sampler` and `mem`. It also uses `with_address` to ensure that the data generated by this sampler is easy to find, in traces of this program.

```

target_trace = {{ }};
data = [100, 101, -99, -100, 99, -101, 100.5, 99.5, -100.5, -99.5];
for_each(range(length(data)), (i) -> {
  target_trace[/datum/$i/normal/] := *data[/$i/];
});
data_addresses = addresses_of(target_trace);

markov_chain_state = {{ }};

trace_choices(
  program          = generate_from_dirichlet_process_mixture,
  inputs           = [length(data)],
  // ensures the initial state has the data
  intervention_trace = target_trace,
  output_trace      = markov_chain_state
);

approximate_inference_update = () ~> {
  single_site_metropolis_hastings_step(
    program = generate_from_dirichlet_process_mixture,
    inputs  = [length(data)],
    trace    = markov_chain_state,
    constraint_addresses = data_addresses
  );
};

repeat(
  times=1000,
  program = approximate_inference_update
);

all_data =
  interpret(
    program = generate_from_dirichlet_process_mixture,
    inputs  = [length(data) + 100], // 100 predictive samples
    intervention_trace = markov_chain_state );
predictive_data = all_data[length(data):end];

```

Figure 42. Metaprob source code for a meta-program that adds an observed dataset, does 1000 steps of lightweight Metropolis-Hastings inference on executions of the program from Figure 33, and then generates 100 samples from the predictive distribution. This meta- program traces the program to obtain an initial state, runs 1000 transitions, then re-executes the original program treating this final trace as source of interventions, to generate samples from the posterior predictive. (Radul and Mansinghka, 2018)

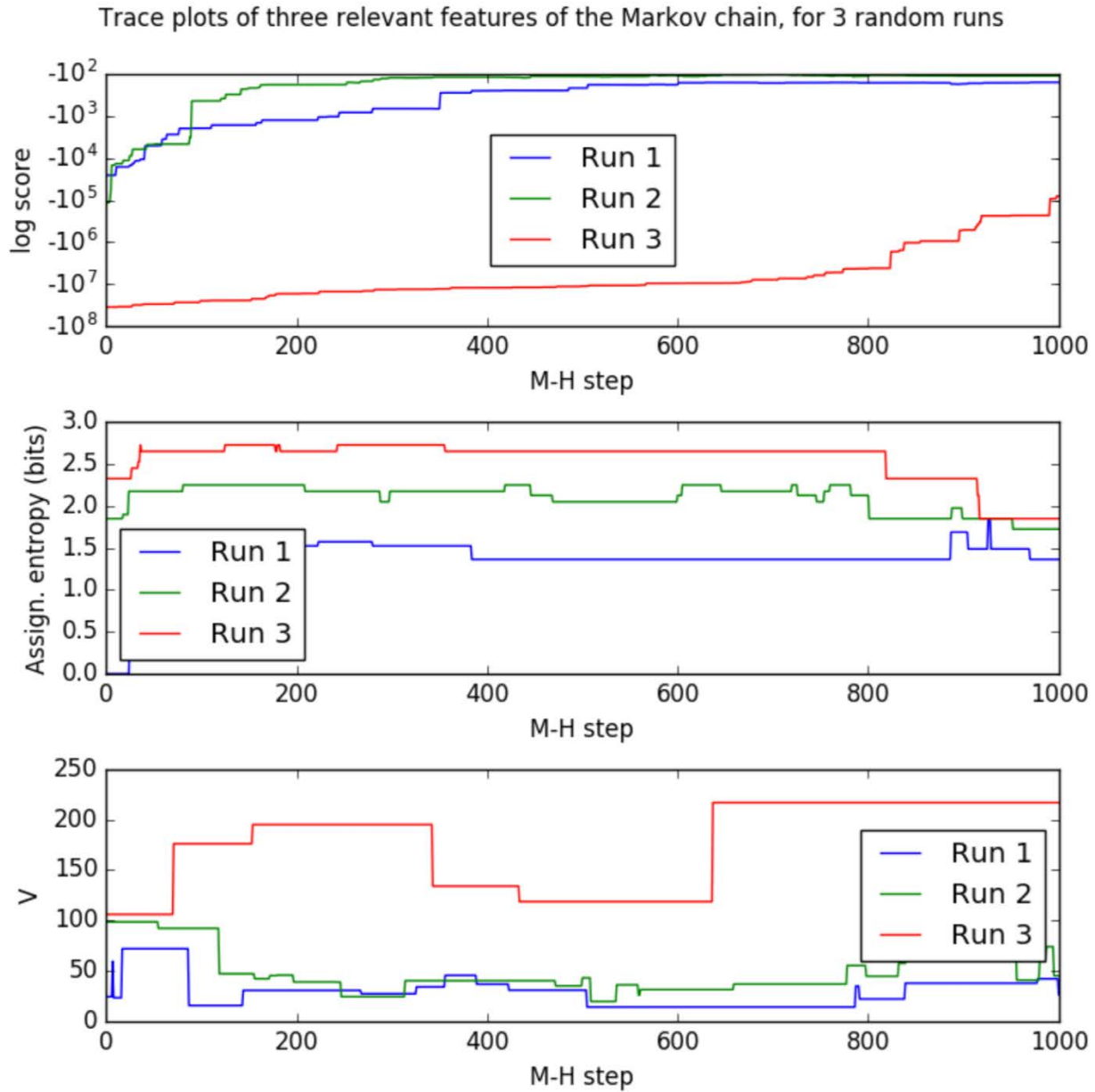


Figure 43. Progress of inference in the Dirichlet process mixture of Gaussians model with hyperparameter inference from Figure 42, above. The top panel shows the evolution of the log probability (density) of the current state. Note that the logarithm is itself displayed on a logarithmic scale. The center panel shows the Shannon entropy of the cluster assignment in bits: 0 corresponds to all points in one cluster, 1 to the points split evenly among two clusters, etc. The bottom panel shows the values taken by the V hyperparameter, which controls the cluster spread to cluster width ratio. (Radul and Mansinghka, 2018)

4.3 Example results drawn from program challenge problems

This section of the report summarizes representative results on (i) the January 2017 Hackathon results on the Gapminder dataset and (ii) Challenge Problem 7: Flu forecasting. Additional program challenge problems and team challenge problems are discussed elsewhere in this report.

4.3.1 January 2017 Hackathon - Gapminder Dataset

The goal of the January 2017 Hackathon was to demonstrate the capabilities of probabilistic programming on a variety of four data analysis tasks: (i) imputation, (ii) modeling the effect of intervention variables on outcome variables, (iii) modeling missing data patterns using CrossCat and BayesDB, and (iv) model criticism and refinement. The original dataset contained roughly 80 parallel time series, with many missing values.

Task 1: Imputation of missing values

Create a joint model of the data and then sample from the joint posterior over the missing values.

For this task, probabilistic programming allowed us to quickly try different modeling assumptions, as well as different assumptions about which parts of the data may be relevant for imputing any given missing value. Our team was able to show results for three approaches: (i) Bayesian time series models fit independently for each variable for each country; (ii) Independent CrossCat models for each year; and (iii) CrossCat models over sets of adjacent years, which can discover which cross-sectional or temporal dependencies are relevant for prediction.

Task 2: Modeling the Effect of Intervention Variables on Outcome Variables

Fit models of the outcome variables as a function of intervention variables (and potential confounding variables as appropriate) to generate causal hypotheses about the effects of intervention variables on outcome variables.

On this task, our team was able to both (i) detect the existence of predictive relationships, as well as (ii) quantify their probable strength. We also hypothesized the existence of latent variables derived from the model structures and then tested whether they could be used to decouple the relationship between a given intervention and outcome variable.

Task 3: Modeling Missing Data Patterns using CrossCat and BayesDB

Determine cases where the missing values are not missing at random.

For this task, data of 170 countries and 51 variables (socioeconomic indicators) over 51 years was provided. Some entries were missing. The purpose of this task is to determine whether the missing values can safely be treated as random, or whether there are values missing not at

random (MNAR). If there were values MNAR, we were asked to fit a model conditioned on the observation of "missingness."

By building a CrossCat generative probabilistic model of "missingness" in 6 cross-sections of time between 1960 to 2010, we were able to expose clusters of variables and clusters of countries, to reveal the rich set of relationships that explain "missingness". Specifically, we identified three apparent causal processes on which the missing data patterns depend in a cross-section of time: (1) geographic/economic properties of the countries, (2) processes related to data collection, and (3) processes related to violent conflict / civil unrest.

1. **Geographic and economic patterns.** The first cluster of missing variables is a cluster in which some -- approximately 1/5 to 3/5 -- of values are missing. We found that this pattern of missingness depends on the geographic and economic category of the country. For example, Western Democracies are more likely to have similar missing values to one another than to developing countries in Africa. For African countries, the converse is true.
2. **Data collection patterns.** The second cluster of variables in 2000 -- with similar clusters observed across years -- is one that contains variables sanitation and water source. The analysis revealed that for almost all countries, a country has both of these missing or neither missing. More rarely does a country have one missing but not the other. The countries included do not have an obvious economic or geographic pattern -- instead, there appears to be some technicality of data collection or analysis that accounts for this pattern of missingness. The interpretation is that these two variables are collected by the same agency or using a similar technology. Corroborating what BayesDB found, a web search shows that these two variables are collected and analyzed together.
3. **Conflict zones.** The third cluster of variables are those variables that, in the vast majority of countries, are "not missing." However, there is a small number of countries that are notable exceptions. Those countries appear to be high conflict zones around the year 2000 or countries that have had governmental transitions. For example, this cluster includes Afghanistan, Iraq, South Sudan, Serbia, and Montenegro. In these conflict zones, more variables are missing than normal including some that are almost always collected.

Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
missing_storm	gdp_per1k	sanitation	invest_foreign_per1k	missing_air_accident	missing_continent
missing_food	energy_per1k	water_source	population_density	food	missing_hiv
missing_sugar	co2_per1	missing_sanitation	epidemic	missing_tsunami	missing_hepatitis
missing_life_expectancy	electricity_per1	missing_water_source	flood	missing_mortality_kid	missing_hiv
agriculture	spending_health_per1	missing_mortality_maternal	drought	missing_completion	immune_hepatitis
missing_invest_foreign_per1k	missing_drought		population	missing_gdp_per1k	missing_immune_hepatitis
missing_surface	missing_aid_received		air_accident	missing_population	mortality_maternal
missing_mcv	it_tel		storm		missing_electricity_per1
missing_labor	invest_domestic_per1k		missing_natural_gas_per1		completion
missing_it_tel	internet		natural_gas_per1		immune_diphtheria
missing_internet	cell_phone		malaria		mcv
missing_infection	computers		hiv		missing_earthquake
missing_immune_diphtheria	missing_professional_birth		surface		gini
missing_computers	professional_birth				missing_gini
missing_co2_per1	missing_malaria				coal_per1
missing_cell_phone	infection				missing_coal_per1
labor	sugar				missing_oil_per1
missing_agriculture	life_expectancy				extreme_temp
missing_population_density	mortality_kid				aid_received
missing_spending_health_per1k	missing_extreme_temp				earthquake
missing_floor	immune_hib				
missing_invest_domestic_per1k	missing_immune_hib				
	missing_physicians				
	physicians				
	missing_immune_tetanus				
	immune_tetanus				
	missing_epidemic				
	broadband				
	missing_broadband				
	oil_per1				

Table 1: Clusters of variables, including raw values and missingness indicators, detected by BayesDB. BayesDB detected several groups of variables that were known to the evaluation team, and found additional relationships as well.

Task 4: Model Criticism and Refinement

Develop a refinement of the model (or completely replace it) to obtain a better model.

For this task, BayesDB was used to (i) study the characteristics and issues of the baseline log-linear model for maternal_mortality in the 3 datasets and (ii) develop and characterize an improved probabilistic model for maternal_mortality, as well as compare its performance to baseline log-linear regression.

4.3.2 Challenge Problem #7 - Flu Problem 2017³

This challenge problem focused on predicting seasonal rates of influenza-like illness (ILI) in 60 sub-populations in the US, ranging in size from the individual counties to the entire country. Data included ILI rate data for each population, as well as three different kinds of covariates data (flu-related tweets, vaccination claims, and weather).

This problem is an instance of a general problem class: jointly modeling hundreds of time series with widely varying, non-stationary dynamics. Multivariate time series data is ubiquitous, arising in domains such as macroeconomics, neuroscience, and public health. Unfortunately, forecasting, imputation, and clustering problems can be difficult to solve when there are tens or hundreds of time series. One challenge in these settings is that the data may reflect underlying processes with widely varying, non-stationary dynamics. Another challenge is that standard approaches based on auto-regressive models, such as AR(p), ARMA, ARIMA, and GARCH models, can become statistically unstable and computationally intensive. Auto-regressive models also require users to know how to choose from a large set of possible parameter settings and appropriately transform the data. Other technical issues that arise include non-linear dependencies, missing data, the presence of outliers, redundant variables, and sparse dependencies

Our probabilistic programming research under PPAML enabled us to develop and apply a domain-general nonparametric Bayesian model for multivariate time series which aims to address some of the above challenges. We implemented Markov chain Monte Carlo algorithms for forecasting, imputation, and clustering based on this model. The modeling approach is based on two extensions to Dirichlet process mixture models. First, we introduce an extension to the Chinese restaurant process that captures temporal dependencies in the generated data. Second, we introduce a hierarchical extension that allows us to cluster time series into groups whose underlying dynamics are modeled and forecast jointly. We used the approach to forecast flu incidence rates in 10 US regions, based on data from the US Center for Disease Control and Prevention. Our approach uses weather and social media time series as covariates.

Quantitative experiments showed that the proposed approach outperforms several Bayesian and nonBayesian baselines, including Facebook's Prophet algorithm, Gaussian process models, and seasonal ARIMA models. We also show competitive imputation performance with state-of-the-art time series imputation techniques such as Amelia II. Two sets of predictions are shown below, for the state of Tennessee and the state of Texas.

³ This section is based on Saad and Mansinghka, 2017.

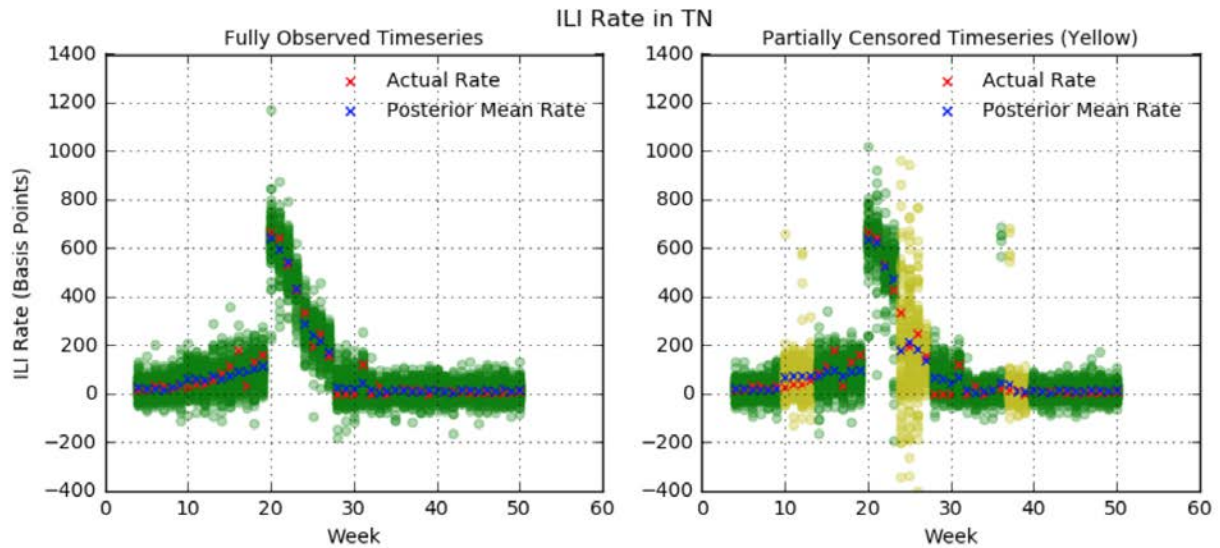


Figure 44. Influenza-like illness rates in Tennessee, including both raw data and forecasts from the probabilistic programs developed for this challenge problem.

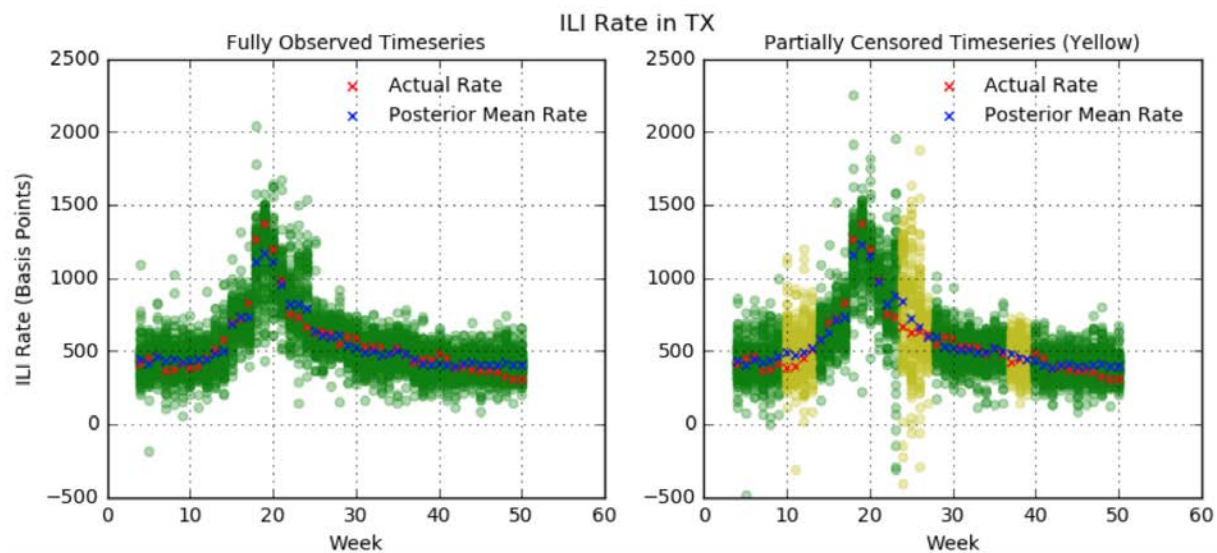


Figure 45. Influenza-like illness rates in Texas, including both raw data and forecasted rates from the probabilistic program developed to solve this challenge problem.

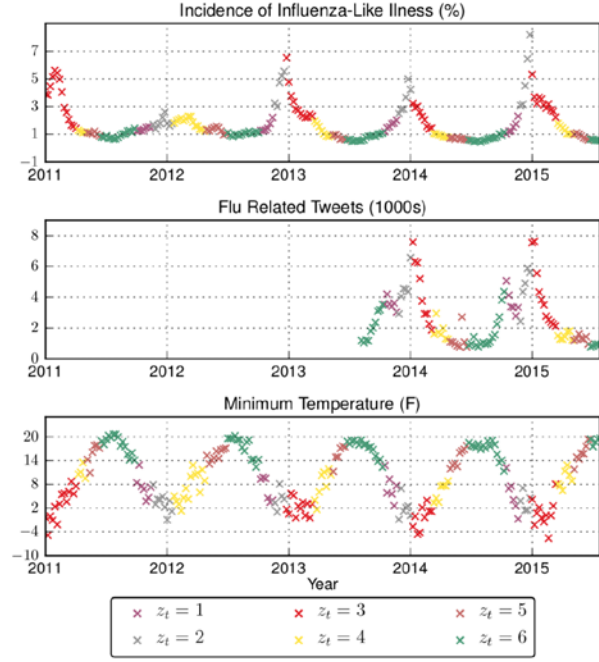
Our team solved this problem by applying a series of different Bayesian model discovery techniques using BayesDB. This included the standard Bayesian model discovery methods, as well as novel methods developed during our PPAML no-cost extension. In these models, the dependencies between time series data sources are mediated by shared cluster assignments, which ensure that all the time series share the same segmentation of the time course into various temporal regimes. For example, in one posterior sample, xix temporal regimes describing the seasonal behavior shared among x_flu , x_tweet , and x_temp are shown: purple, gray, and red

are the pre-peak rise, peak, and post-peak decline during the flu season; yellow, brown, and green represent the rebound in between successive seasons. In 2012, the model reports absence of the red post-peak regime, reflecting the season's mild flu peak.

1. Sample concentration parameter of CRP
 $\alpha \sim \text{Gamma}(1,1)$
2. Sample model hyperparameters ($n = 1, 2, \dots, N$)
 $\lambda_G^n \sim H_G^n$
 $\lambda_F^n \sim H_F^n$
3. Sample distribution parameters of F ($n = 1, 2, \dots, N$)
 $\theta_1^n, \theta_2^n, \dots \stackrel{\text{iid}}{\sim} \pi_\Theta(\cdot | \lambda_F^n)$
4. Assume first p values are known ($n = 1, 2, \dots, N$)
 $\mathbf{x}_{-p+1:0}^n \sim (\mathbf{x}_{-p+1}^n, \dots, \mathbf{x}_0^n)$
5. Sample time series observations ($t = 1, 2, \dots$)
 - 5.1 Sample temporal cluster assignment z_t

$$\Pr[z_t = k | \mathbf{z}_{1:t-1}, \mathbf{x}_{t-p:t-1}^{1:N}, \alpha] \propto \text{CRP}(k | \alpha, \mathbf{z}_{1:t-1}) \prod_{n=1}^N G(\mathbf{x}_{t-p:t-1}^n; D_{tk}^n, \lambda^n)$$

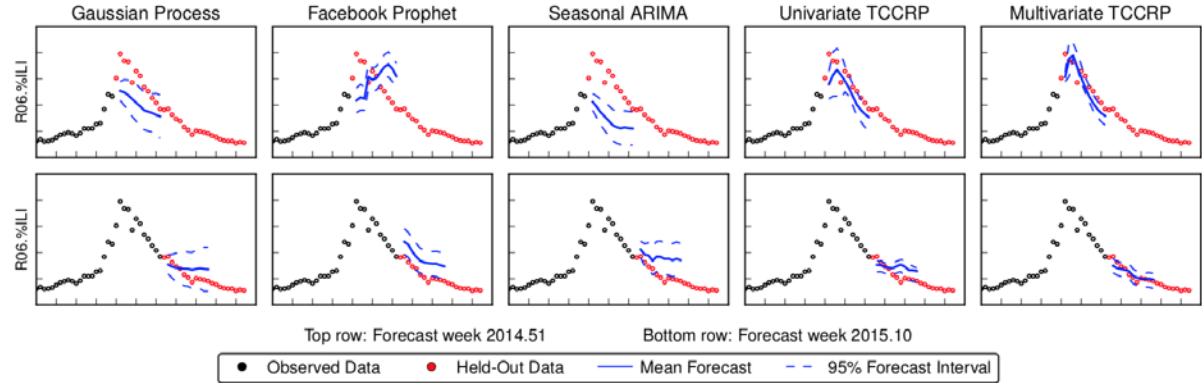
where $D_{tk}^n := \{\mathbf{x}_{t'-p:t'-1}^n | z_{t'} = k, 1 \leq t' < t\}$
and $k = 1, \dots, \max(\mathbf{z}_{1:t-1}) + 1$
 - 5.2 Sample data x_t^n ($n = 1, 2, \dots, N$)
 $x_t^n | z_t, \{\theta_k^n\} \sim F(\cdot | \theta_{z_t}^n)$



(a) Generative process for the multivariate TCCRP mixture (b) Discovering flu season dynamics with the TCCRP

Figure 46. (a) Hierarchical Bayesian model for the joint distribution of N dependent time series $\{\mathbf{x}_n\}$, using a multivariate temporally-coupled CRP mixture. (b) example latent structure for data from the Flu challenge problem, showing how the discovered time series models break down their put data into clusters. (Saad and Mansinghka, 2017).

(a) Forecasting the 2015 flu season in US Region 6. In these two representative weeks (top: week 2014.51, bottom: week 2015.10), the multivariate TCCRP mixture accurately forecasts both the pre- and post-peak seasonal dynamics (right-most column), whereas state-of-the-art baselines demonstrate inaccurate forecasts and/or miscalibrated uncertainties.



(b) Mean absolute flu prediction error for ten forecast horizons (in weeks) averaged over 10 populations

	$h = 1$	$h = 2$	$h = 3$	$h = 4$	$h = 5$	$h = 6$	$h = 7$	$h = 8$	$h = 9$	$h = 10$
Constant	0.53 _(0.04)	0.60 _(0.03)	0.66 _(0.03)	0.71 _(0.03)	0.75 _(0.02)	0.79 _(0.02)	0.82 _(0.02)	0.85 _(0.02)	0.87 _(0.02)	0.89 _(0.02)
Linear Extrapolation	0.65 _(0.06)	0.79 _(0.05)	0.93 _(0.05)	1.08 _(0.05)	1.24 _(0.05)	1.39 _(0.05)	1.55 _(0.05)	1.70 _(0.05)	1.86 _(0.05)	2.01 _(0.05)
GP (SE+PER+WN)	0.53 _(0.04)	0.60 _(0.03)	0.66 _(0.03)	0.71 _(0.03)	0.75 _(0.02)	0.79 _(0.02)	0.82 _(0.02)	0.85 _(0.02)	0.87 _(0.02)	0.89 _(0.02)
GP (SE×PER+WN)	0.50 _(0.04)	0.57 _(0.03)	0.62 _(0.03)	0.67 _(0.02)	0.71 _(0.02)	0.74 _(0.02)	0.78 _(0.02)	0.81 _(0.02)	0.84 _(0.02)	0.86 _(0.02)
Facebook Prophet [31]	0.83 _(0.04)	0.84 _(0.03)	0.85 _(0.02)	0.85 _(0.02)	0.85 _(0.02)	0.86 _(0.02)	0.86 _(0.02)	0.87 _(0.02)	0.87 _(0.01)	0.87 _(0.01)
Seasonal ARIMA [13]	0.64 _(0.04)	0.76 _(0.03)	0.84 _(0.03)	0.92 _(0.03)	0.98 _(0.03)	1.04 _(0.02)	1.08 _(0.02)	1.13 _(0.02)	1.16 _(0.02)	1.19 _(0.02)
Univariate TCCRP	0.54 _(0.04)	0.58 _(0.03)	0.62 _(0.02)	0.67 _(0.02)	0.71 _(0.02)	0.76 _(0.02)	0.80 _(0.02)	0.83 _(0.02)	0.86 _(0.02)	0.89 _(0.02)
Multivariate TCCRP	0.46_(0.03)	0.49_(0.02)	0.51_(0.02)	0.53_(0.02)	0.56_(0.02)	0.58_(0.02)	0.59_(0.01)	0.61_(0.01)	0.62_(0.01)	0.64_(0.01)

Figure 47. Quantitative evaluation of forecasting performance using several simple and state-of-the-art baselines (a) Examples of pre and post-the peak forecasts in US Region 6. (b) Mean prediction errors and one standard error for short, medium, and long-term forecast horizons averaged over Regions 1 through 10. Predictive improvement of the multivariate TCCRP mixture over baseline methods is especially apparent at higher forecast horizons (Saad & Mansinghka, 2017).

4.4 Example results for our team challenge problem of analyzing a database of Earth satellites using BayesDB.

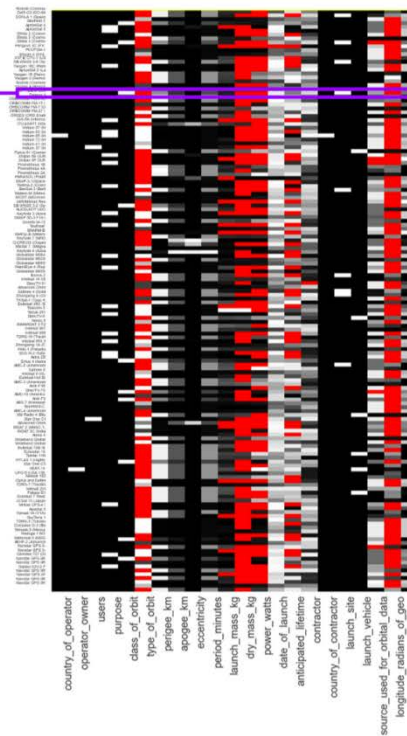
One of the team challenge problems over the course of the PPAML program was focused on providing computer-aided judgment for ‘medium data’ problems, and whether it was possible to make statistical inference broadly accessible to non-statisticians without sacrificing mathematical rigor or inference quality. We tested these capabilities by applying BayesDB to a public database of Earth satellites and measuring the runtime and accuracy of a broad class of queries.

This section outlines a case study applying compositional generative population models (CGPMs) in BayesDB to a population of satellites maintained by the Union of Concerned Scientists. The dataset contains 1163 entries, and each satellite has 23 numerical and categorical features such as its material, functional, physical, orbital and economic characteristics. We construct a hybrid CGPM using an MML metamodel definition which combines (i) a classical physics model written as a probabilistic program in VentureScript, (ii) a random forest to classify a nominal variable, (iii) an ordinary least squares regressor to predict a numerical variable, and (iv) principal component analysis on the real-valued features of the satellites. These CGPMs allow us to identify satellites that probably violate their orbital mechanics, accurately infer missing values of anticipated lifetime, and visualize the dataset by projecting the satellite features into two dimensions.

Data for Compass M4

	0
Name	Compass M4 (Beidou 2-13)
Country_of_Operator	China (PR)
Operator_Owner	Chinese Defense Ministry
Users	Military
Purpose	Navigation/Global Positioning
Class_of_Orbit	MEO
Type_of_Orbit	NaN
Perigee_km	21452
Apogee_km	21603
Eccentricity	0.00271
Period_minutes	773.21
Launch_Mass_kg	2200
Dry_Mass_kg	NaN
Power_watts	NaN
Date_of_Launch	41027
Anticipated_Lifetime	8
Contractor	Space Technology Research Institute (part of C...
Country_of_Contractor	China (PR)
Launch_Site	Xichang Satellite Launch Center
Launch_Vehicle	Long March 3B
Source_Used_for_Orbital_Data	ZARYA
longitude_radians_of_geo	NaN
Inclination_radians	0.961676

Columns: descriptive variables



Rows:
Satellites

Red represents missing data

Figure 48. Example application: Earth-orbiting satellites. The right panel shows a schematic of the dataset, released publicly by the Union of Concerned Scientists. The left panel shows a single example record corresponding to a Chinese Defense Ministry satellite.

```
%mm1 CREATE TABLE satellites
... FROM satellites.csv;

%mm1 CREATE POPULATION SCHEMA FOR satellites
... FROM satellites_data
... WITH SCHEMA (
...     GUESS STATYPES FOR (*);
... );

%mm1 INITIALIZE 64 POPULATION MODELS FOR
... satellites USING MODELING TACTIC crosscat;

%mm1 IMPROVE MODELS FOR satellites FOR 4 MINUTES;
```

Figure 49. Example probabilistic code written in BayesDB for Bayesian model discovery from this satellites database.

```

CREATE TABLE satellites_ucs FROM 'satellites.csv'
.nullify satellites_ucs 'NaN'
CREATE POPULATION satellites FOR satellites_ucs WITH SCHEMA (
  IGNORE Name;
  GUESS STATTYPES FOR *;
  MODEL longitude_radians_of_geo, inclination_radians AS CYCLIC;
);

CREATE POPULATION SCHEMA sat_hybrid FOR satellites WITH BASELINE crosscat(
  OVERRIDE GENERATIVE MODEL
  FOR launch_mass_kg, dry_mass_kg, power_watts, perigee_km, apogee_km
  EXPOSE pc1 NUMERICAL, pc2 NUMERICAL
  USING factor_analysis(L=2);
  OVERRIDE GENERATIVE MODEL
  FOR anticipated_lifetime
  GIVEN date_of_launch, power_watts, apogee_km, perigee_km, dry_mass_kg, class_of_orbit
  USING linear_regression;
  OVERRIDE GENERATIVE MODEL FOR type_of_orbit
  GIVEN apogee_km, perigee_km, period_minutes, users, class_of_orbit
  USING random_forest(k=7);
);

INITIALIZE 16 POPULATION MODELS FOR satellites;
IMPROVE MODELS FOR 4 MINUTES;

```

Customize statistical types

Factor analysis with exposed factors

Linear regression to predict lifetime

Random forests to classify orbit types

Figure 50. Example probabilistic code written in BayesDB for improving over baseline Bayesian model discovery by integrating custom models. This demonstrates a unique capability of BayesDB: combining baseline probabilistic programs that were synthesized by an automatic mechanism with custom probabilistic code written by an end user.

"What are the satellites in geosynchronous orbit with the most unlikely orbital periods?"

```

ESTIMATE name, class_of_orbit, period_minutes, PROBABILITY OF period_minutes
FROM satellites
WHERE class_of_orbit = GEO
ORDER BY PROBABILITY OF period_minutes ASCENDING LIMIT 10

```

	Name	Class_of_Orbit	Period_minutes	Relative Probability of Period	
0	AEHF-3 (Advanced Extremely High Frequency sate...	GEO	1306.29	0.001295	
1	AEHF-2 (Advanced Extremely High Frequency sate...	GEO	1306.29	0.001295	
2	DSP 20 (USA 149) (Defense Support Program)	GEO	142.08	0.002638	Our explanation: off by 10x; not an outlier
3	Intelsat 903	GEO	1436.16	0.003249	
4	BSAT-3B	GEO	1365.61	0.003418	
5	Intelsat 902	GEO	1436.10	0.003443	
6	SDS III-6 (Satellite Data System) NRO L-27, Gr...	GEO	14.36	0.003735	
7	Advanced Orion 6 (NRO L-15, USA 237)	GEO	23.94	0.003863	Our explanation: Meant 24 hours not 24 minutes
8	SDS III-7 (Satellite Data System) NRO L-38, Dr...	GEO	23.94	0.003863	
9	QZS-1 (Quazi-Zenith Satellite System, Michibiki)	GEO	1436.00	0.004522	

Figure 51. An example query in BayesDB's Bayesian Query Language (BQL) that identifies probably anomalous satellites, along with example results.

Traditional databases protect consumers of data from “having to know how the data is organized in the machine” Codd (1970) and provide automated data representations and retrieval algorithms that perform well enough for a broad class of applications. Although this abstraction barrier is only imperfectly achieved, it has proved useful enough to serve as the basis of multiple generations of software and data systems. This decoupling of task specification from implementation made it possible to improve performance and reliability — of individual

database indexes, and in some cases of entire database systems — without needing to notify end users. It also created a simple conceptual vocabulary and query language for data management and data processing that spread far farther than the systems programming knowledge needed to implement it.

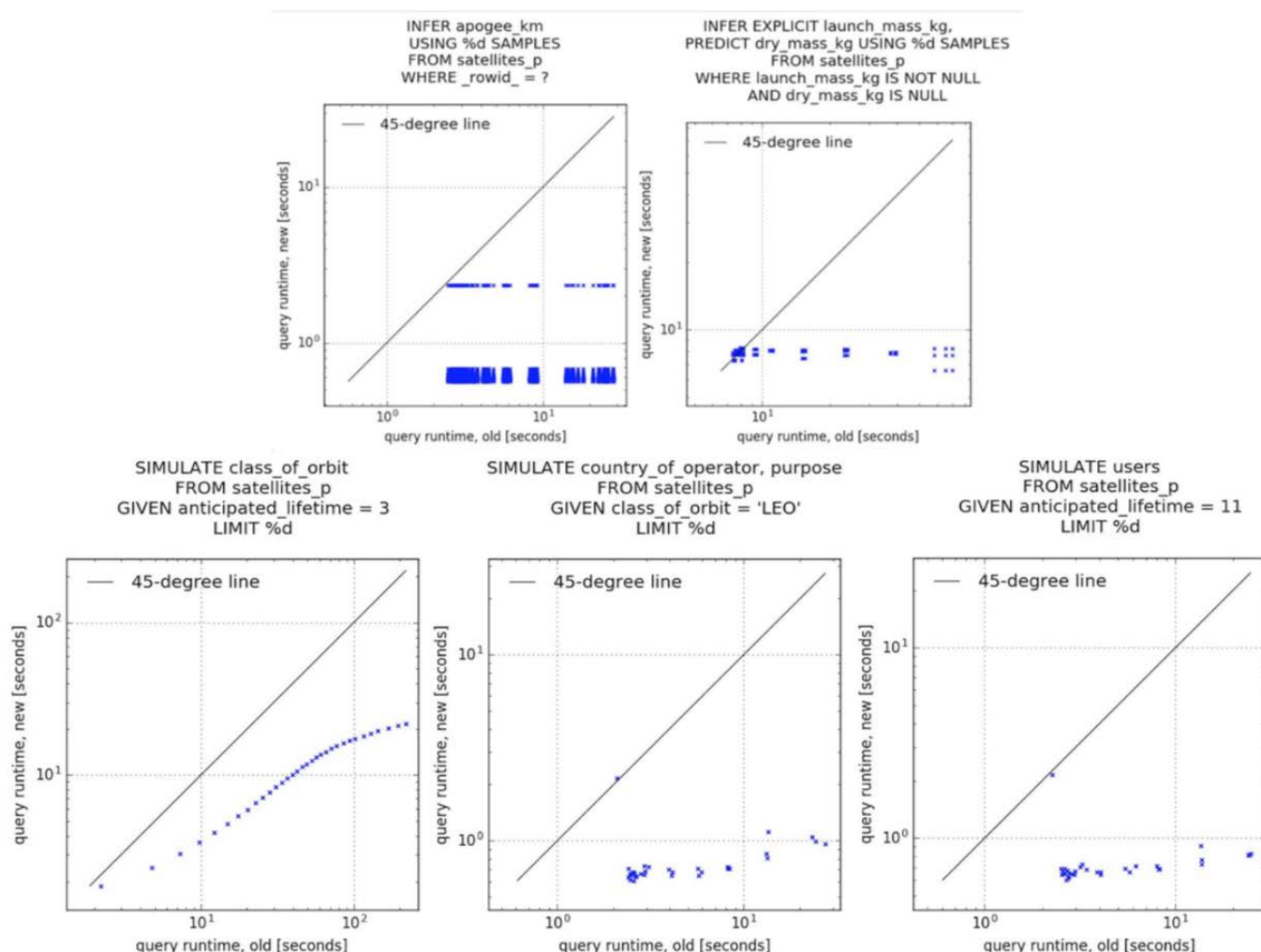


Figure 52. Query performance results on the Satellites database, showing improvement in runtime for multiple queries between our first submission and second submission. These results quantitatively show that we were able to significantly improve the speed of BayesDB over the course of the PPAML program.

4.5 Example results showing how probabilistic programming can be used to reimplement the "Automatic Statistician" in under 70 lines of code.

Another challenge problem we focused on, described in our initial proposal, is showing that probabilistic programming can be used to re-implement state-of-the-art techniques for discovering symbolic structure in data. The system we chose to reimplement is called the

Automated Statistician. This system was developed at Cambridge University is described as follows on its website:

"Making sense of data is one of the great challenges of the information age we live in. While it is becoming easier to collect and store all kinds of data, from personal medical data, to scientific data, to public data, and commercial data, there are relatively few people trained in the statistical and machine learning methods required to test hypotheses, make predictions, and otherwise create interpretable knowledge from this data. The Automatic Statistician project aims to build an artificial intelligence for data science, helping people make sense of their data."

Kevin P. Murphy, Senior Research Scientist at Google says: "In recent years, machine learning has made tremendous progress in developing models that can accurately predict future data. However, there are still several obstacles in the way of its more widespread use in the data sciences. The first problem is that current Machine Learning (ML) methods still require considerable human expertise in devising appropriate features and models. The second problem is that the output of current methods, while accurate, is often hard to understand, which makes it hard to trust. The "automatic statistician" project from Cambridge aims to address both problems, by using Bayesian model selection strategies to automatically choose good models / features, and to interpret the resulting fit in easy-to-understand ways, in terms of human readable, automatically generated reports."

The figure below gives an example input and output from the Automated Statistician, applied to a database of airline passenger counts spanning multiple years.

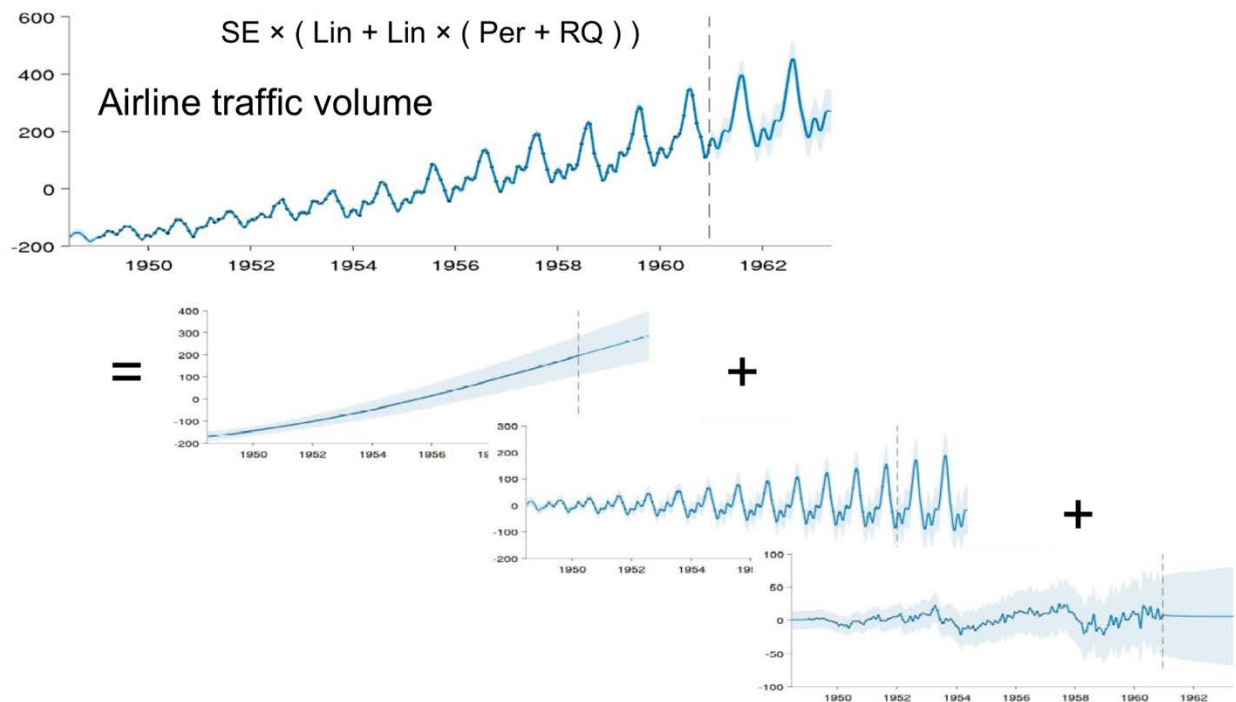


Figure 53. The passenger counts can be decomposed into a linearly increasing trend, a periodic overlay (with amplitude that increases over time), and a set of local stochastic deviations from the main trend.

The core probabilistic inference technique behind these AI capabilities are described as follows in (Duvenaud et al., 2013), the original paper that led to the Automatic Statistician:

Abstract: *Despite its importance, choosing the structural form of the kernel in nonparametric regression remains a black art. We define a space of kernel structures which are built compositionally by adding and multiplying a small number of base kernels. We present a method for searching over this space of structures which mirrors the scientific discovery process. The learned structures can often decompose functions into interpretable components and enable long-range extrapolation on time-series datasets. Our structure search method outperforms many widely used kernels and kernel combination methods on a variety of prediction tasks.*

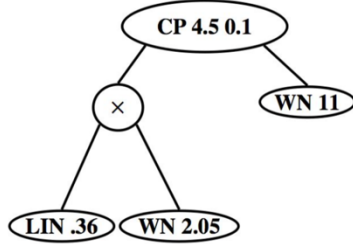
Over the course of PPAML, we showed how to reimplement and extend the Automatic Statistician using the Venture probabilistic programming language. The key technical ideas are to (i) represent models using abstract syntax trees for a domain-specific probabilistic language, and (ii) represent the time series model prior, likelihood, and search strategy using probabilistic programs in a sufficiently expressive language. The final probabilistic program is written in under 70 lines of probabilistic code in Venture. Below, we demonstrate an application to time series clustering that involves a non-parametric extension to ABCD, experiments for interpolation and extrapolation on real-world econometric data, and improvements in accuracy over both non-parametric and standard regression baselines.

We formulated structure discovery as a form of “probabilistic program synthesis”. The key idea is to represent probabilistic models using abstract syntax trees (ASTs) for a domain-specific language, and then use probabilistic programs to specify the AST prior, model likelihood, and search strategy over models given observed data.

Several recent projects have applied probabilistic programming techniques to Gaussian process time series. Schaechtle et al. (2015) embed GPs into Venture with fully Bayesian learning over a limited class of covariance structures with a heuristic prior. Tong and Choi (2016) describe a technique for learning GP covariance structures using a relational variant of ABCD, and then compile the models into Stan (2017). However, probabilistic programming is only used for prediction, not for structure learning or for hyperparameter inference.

The contributions of the ABCD approach we took are as follows. First, we formulate ABCD as probabilistic program synthesis. Second, our implementation supports combinations of gradient-based search for hyperparameters, and Metropolis-Hastings sampling for structure and hyperparameters. Third, we showed competitive performance on extrapolation and interpolation tasks from real-world data against several baselines. Fourth, we showed that 10 lines of code are sufficient to extend ABCD into a nonparametric Bayesian clustering technique that identifies time series which share covariance structure.

Data structure representing probabilistic program



Probabilistic program implementing Gaussian process model

```

assume covariance_kernel =
  gp_cov_cp(
    4.5, .1,
    gp_cov_product(
      gp_cov_linear(.36),
      gp_cov_scale(
        2.05, gp_cov_delta)),
    gp_cov_scale(
      11, gp_cov_delta));
assume gp = make_gp(
  gp_mean_const(0.),
  covariance_kernel);

```

Three simulated realizations

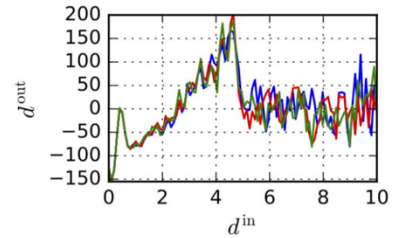


Figure 54. Executing synthesized model programs to produce Gaussian process datasets. Left: A symbolic structure generated by the AST prior. Center: Equivalent source code of the Venture GP model program, produced by the AST interpreter. Right: Executions of the model program probed with inputs in the region $[0, 10]$, which outputs datasets of GP time series. (Schaechtle et al., 2017).

Generating hyperparameters

Choosing kernel types

Choosing composition rules

Generating probabilistic program source code

```

1  assume tree_root = () -> {1};
2
3  assume get_hyper_prior ~ mem((node_index) -> {
4    // Gradient-safe exponential prior.
5    -log_logistic(log_odds_uniform()) #hypers:node_index
6  });
7
8  assume choose_primitive = (node) -> {
9    base_kernel ~ categorical(simplex(.2, .2, .2, .2, .2),
10     ["WN", "C", "LIN", "SE", "PER"]) #structure:pair("base_kernel", node);
11    cond(
12      (base_kernel == "WN") ([["WN", get_hyper_prior(pair("WN", node))]]),
13      (base_kernel == "C") ([["C", get_hyper_prior(pair("C", node))]]),
14      (base_kernel == "LIN") ([["LIN", get_hyper_prior(pair("LIN", node))]]),
15      (base_kernel == "SE") ([["SE", .01 + get_hyper_prior(pair("SE", node))]]),
16      (base_kernel == "PER") ([["PER",
17        .01 + get_hyper_prior(pair("PER_l", node)),
18        .01 + get_hyper_prior(pair("PER_t", node))]
19    ]))
20  });
21
22  assume choose_operator = mem((node) -> {
23    operator_symbol ~ categorical(simplex(0.45, 0.45, 0.1),
24     ["+", "*", "CP"]) #structure:pair("operator", node);
25    if (operator_symbol == "CP") {
26      [operator_symbol, get_hyper_prior(pair("CP", node))]
27    } else {
28      operator_symbol
29    }
30  });
31
32  assume generate_random_program = mem((node) -> {
33    if (flip(.3) #structure:pair("branch", node)) {
34      operator ~ choose_operator(node);
35      [operator, generate_random_program(2 * node), generate_random_program(2 * node + 1)]
36    } else {
37      choose_primitive(node)
38    }
39  });

```

Figure 55. Synthesis model: AST prior G - Structure discovery in time series via probabilistic program synthesis. (Schaechtle et al., 2017). This fragment of probabilistic code shows how to define a generative model over probabilistic programs.

Defining the
model and
loading data

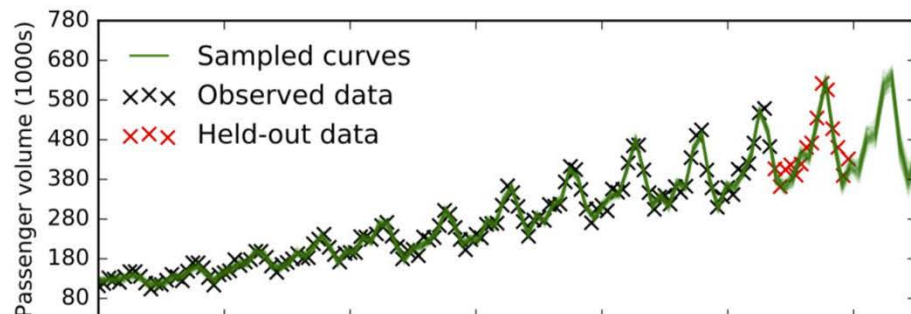
```
assume source ~ generate_random_program(tree_root());
assume gp_executable = produce_executable(source);
define xs = get_data_xs("./data.csv");
define ys = get_data_ys("./data.csv");
observe gp_executable({xs}) = ys;
```

Custom
inference
strategy

```
for_each(arange(T), (_) -> {
  infer gradient(
    minimal_subproblem(/?hypers), steps=100);
  infer resimulate(
    minimal_subproblem(one(/?structure)), steps=100)})
```

Figure 56. Source code for the Automatic Statistician in Venture: data observation program (top) and synthesis inference strategy: MH + Gradients (bottom) (Schaehtle et al., 2017). This code shows how to load data and use a custom inference strategy to solve the problem of discovering symbolic structure in time series data.

Airline
passenger
volume
data



Solar
irradiance
data

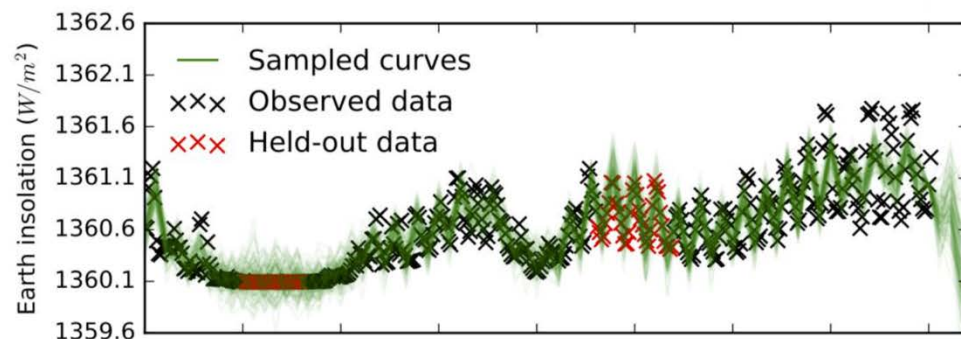


Figure 57. Qualitative results for extrapolation and interpolation tasks. The figure above shows extrapolation performance on a dataset of airline passenger volume between 1949 and 1960. The probabilistic program detects the linear trend with periodic variation, leading to very accurate predictions. In contrast, Bayesian linear regression is forced to treat such structural effects as unmodeled noise (Schaehtle et al., 2017).

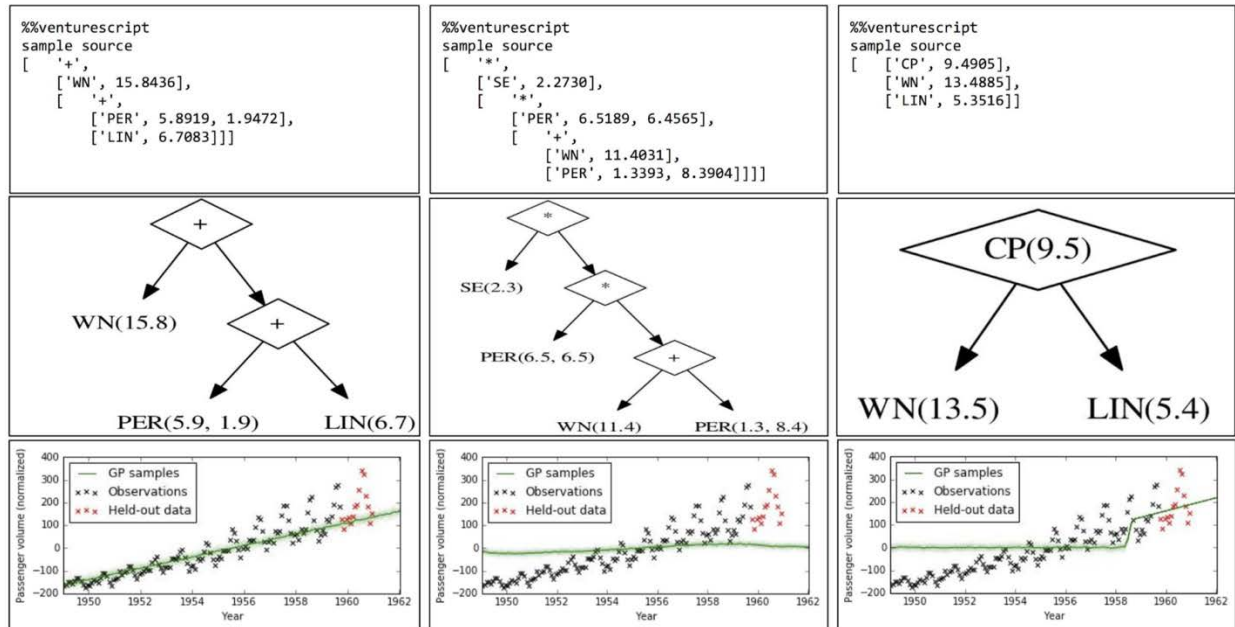


Figure 58. Samples from prior for Airline example. Top: Domain specific language representation of three symbolic structures generated by the AST prior. Middle: tree-representation of said symbolic structures. Bottom: executions of the model program (green) probed with in regimes where data was observed (black data points) and in regimes where data was held out (red data points). The executions of the model programs resulting from the three different symbolic structures model different possible dynamics in the data. The left program represents a linear trend with a smooth and slightly periodic component plus small white noise. The center programs models a smooth curve. The right program implements a changepoint model.

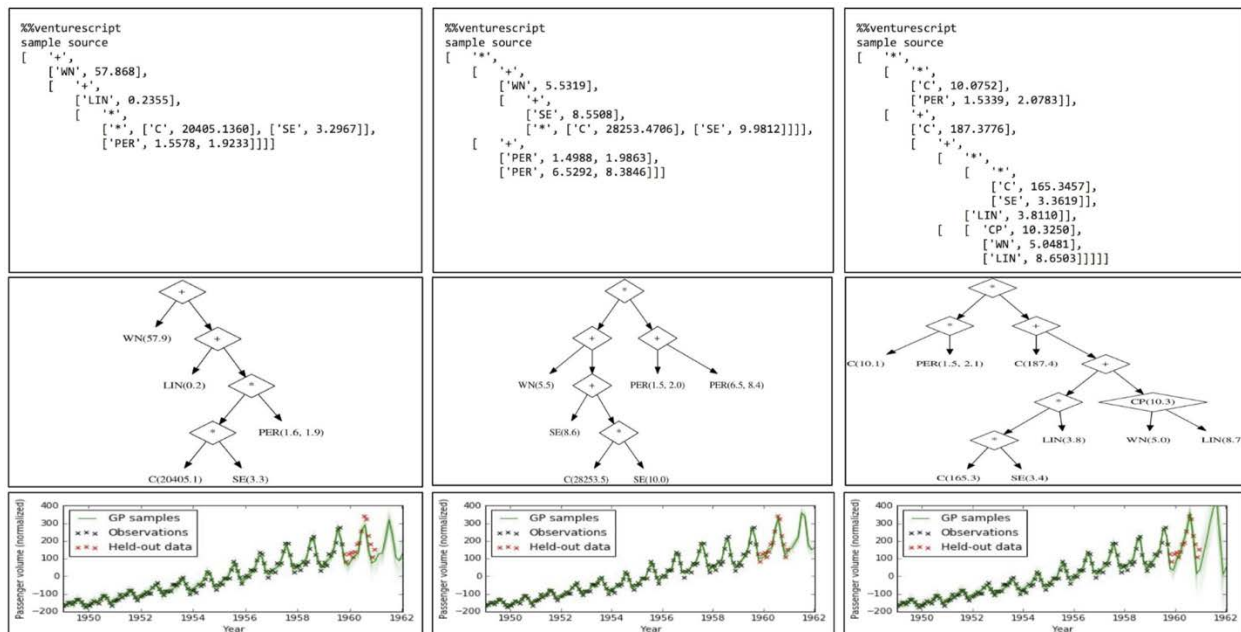


Figure 59. Samples for approximate posterior for Airline example. Top: Domain specific language representation of three symbolic structures generated by the approximate AST posterior. Middle: tree-representation of said symbolic structures. Bottom: executions of the model program (green) probed with in regimes where data was observed (black data points) and in regimes where data was held out (red data points). A comparison of the executions with observed and held out data indicates good inter- and extrapolation performance for the resulting programs given all of the three sampled structures.

	Probabilistic program synthesis	ABCD
Main system	69	4,166
Gaussian process libraries	2,164 (Venture gpmm)	13,945 (GPML Toolbox)
Generic inference implementation	1,887 (Venture)	—

Figure 60. Our probabilistic program implements functionality from the “Automated Statistician” but using ~100x fewer lines of code. (Schaehtle et al., 2017).

Above, we have described and implemented a framework for time series structure discovery using probabilistic program synthesis. We also have assessed efficacy of the approach on synthetic and real-world experiments, and demonstrated improvements in model discovery, extensibility, and predictive accuracy. It seems promising to apply probabilistic program synthesis to several other settings, such as fully-Bayesian search in compositional generative grammars for other model classes (Grosse et al., 2012), or Bayes net structure learning with structured priors (Mansinghka et al., 2006). We hope the formalisms in herein encourage broader use of probabilistic programming techniques to learn symbolic structures in other applied domains.

4.6 Example results showing how probabilistic programming can be used to solve problems of visual scene understanding.

Recent progress on probabilistic modeling and statistical learning, coupled with the availability of large training datasets, has led to remarkable progress in computer vision. Generative probabilistic models, or “analysis-by-synthesis” approaches, can capture rich scene structure but have been less widely applied than their discriminative counterparts, as they often require considerable problem-specific engineering in modeling and inference, and inference is typically seen as requiring slow, hypothesize-and-test Monte Carlo methods. In this section, we describe how Picture (Kulkarni et al, 2015[1]), a probabilistic programming language for scene understanding that allows researchers to express complex generative vision models, while automatically solving them using fast general-purpose inference machinery.

Picture provides a stochastic scene language that can express generative models for arbitrary 2D/3D scenes, as well as a hierarchy of representation layers for comparing scene hypotheses with observed images by matching not simply pixels, but also more abstract features (e.g., contours, deep neural network activations). Inference can flexibly integrate advanced Monte Carlo strategies with fast bottom-up data-driven methods. Thus both representations and inference strategies can build directly on progress in discriminatively trained systems to make generative vision more robust and efficient. We used Picture to write programs for 3D face analysis, 3D human pose estimation, and 3D object reconstruction – each competitive with specially engineered baselines.

Probabilistic scene understanding systems aim to produce high-probability descriptions of scenes conditioned on observed images or videos, typically either via discriminatively trained models or generative models in an “analysis by synthesis” framework. Discriminative approaches lend themselves to fast, bottom-up inference methods and relatively knowledge-free, data-intensive training regimes, and have been remarkably successful on many recognition problems (Felzenszwalb et al., 2010; Krizhevsky et al., 2012; LuCun and Bengio, 1995; Lowe 2004). Generative approaches hold out the promise of analyzing complex scenes more richly and flexibly (Grenander 1993; Grenander, Chow, and Keenan, 1991; Zhao and Zhu, 2011; Del Pero et al., 2013; Jampani et al., 2014; Loper and Black, 2014; Mansinghka et al., 2013[9]; Hinton, Osindero, and Teh, 2006; Jin and Geman, 2006), but have been less widely embraced for two main reasons: Inference typically depends on slower forms of approximate inference, and both model-building and inference can involve considerable problem-specific engineering to obtain robust and reliable results. These factors make it difficult to develop simple variations on state-of-the-art models, to thoroughly explore the many possible combinations of modeling, representation, and inference strategies, or to richly integrate complementary discriminative and generative modeling approaches to the same problem. More generally, to handle increasingly realistic scenes, generative approaches will have to scale not just with respect to data size but also with respect to model and scene complexity. This scaling will arguably require general-purpose frameworks to compose, extend and automatically perform inference in complex structured generative models – tools that for the most part do not yet exist.

One of the programming languages developed over the course of the PPAML project was Picture, which aims to provide a common representation language and inference engine suitable for a broad class of generative scene perception problems. We see probabilistic programming as key to realizing the promise of “vision as inverse graphics”. Generative models can be represented via stochastic code that samples hypothesized scenes and generates images given those scenes. Rich deterministic and stochastic data structures can express complex 3D scenes that are difficult to manually specify. Multiple representation and inference strategies are specifically designed to address the main perceived limitations of generative approaches to vision. Instead of requiring photo-realistic generative models with pixel-level matching to images, we can compare hypothesized scenes to observations using a hierarchy of more abstract image representations such as contours, discriminatively trained part-based skeletons, or deep neural network features. Available Markov Chain Monte Carlo (MCMC) inference algorithms include not only traditional Metropolis-Hastings, but also more advanced techniques for inference in high-dimensional continuous spaces, such as elliptical slice sampling, and Hamiltonian Monte Carlo which can exploit the gradients of automatically differentiable renderers. These top-down inference approaches are integrated with bottom-up and automatically constructed data-driven proposals, which can dramatically accelerate inference by eliminating most of the “burn in” time of traditional samplers and enabling rapid mode-switching. We demonstrate Picture on three challenging vision problems: inferring the 3D shape and detailed appearance of faces, the 3D pose of articulated human bodies, and the 3D shape of medially-symmetric objects. The vast majority of code for image modeling and inference is reusable across these and many other tasks. We show that Picture yields performance competitive with optimized baselines on each of these benchmark tasks.


```

function PROGRAM(MU, PC, EV, VERTEX_ORDER)
# Scene Language: Stochastic Scene Gen
face=Dict();shape = []; texture = [];
for S in ["shape", "texture"]
  for p in ["nose", "eyes", "outline", "lips"]
    coeff = MvNormal(0,1,1,99)
    face[S][p] = MU[S][p]+PC[S][p].*(coeff.*EV[S][p])
  end
end
shape=face["shape"][:]; tex=face["texture"][:];
camera = Uniform(-1,1,1,2); light = Uniform(-1,1,1,2)

# Approximate Renderer
rendered_img= MeshRenderer(shape,tex,light,camera)

# Representation Layer
ren_fts = getFeatures("CNN_Conv6", rendered_img)

# Comparator
#Using Pixel as Summary Statistics
observe(MvNormal(0,0.01), rendered_img-obs_img)
#Using CNN last conv layer as Summary Statistics
observe(MvNormal(0,10), ren_fts-obs_cnn)
end

global obs_img = imread("test.png")
global obs_cnn = getFeatures("CNN_Conv6", img)
#Load args from file
TR = trace(PROGRAM,args=[MU,PC,EV,VERTEX_ORDER])
# Data-Driven Learning
learn_datadriven_proposals(TR,100000,"CNN_Conv6")
load_proposals(TR)
# Inference
infer(TR,CB,20,["DATA-DRIVEN"])
infer(TR,CB,200,["ELLIPTICAL"])

```

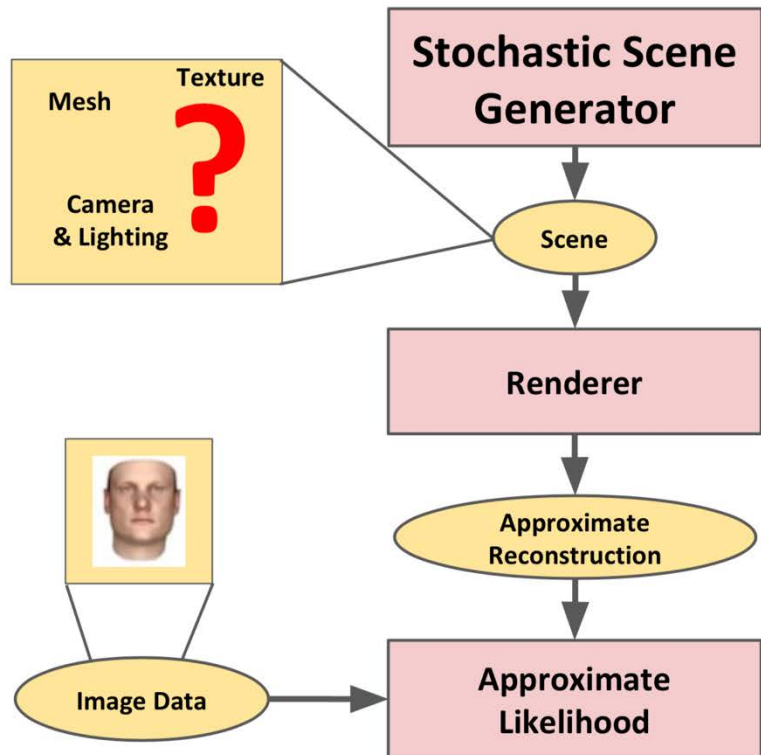


Figure 61. Picture code [left] for the 3D face application, along with a schematic description of the dependencies within that Picture program [right]. (Kulkarni et al., 2015).

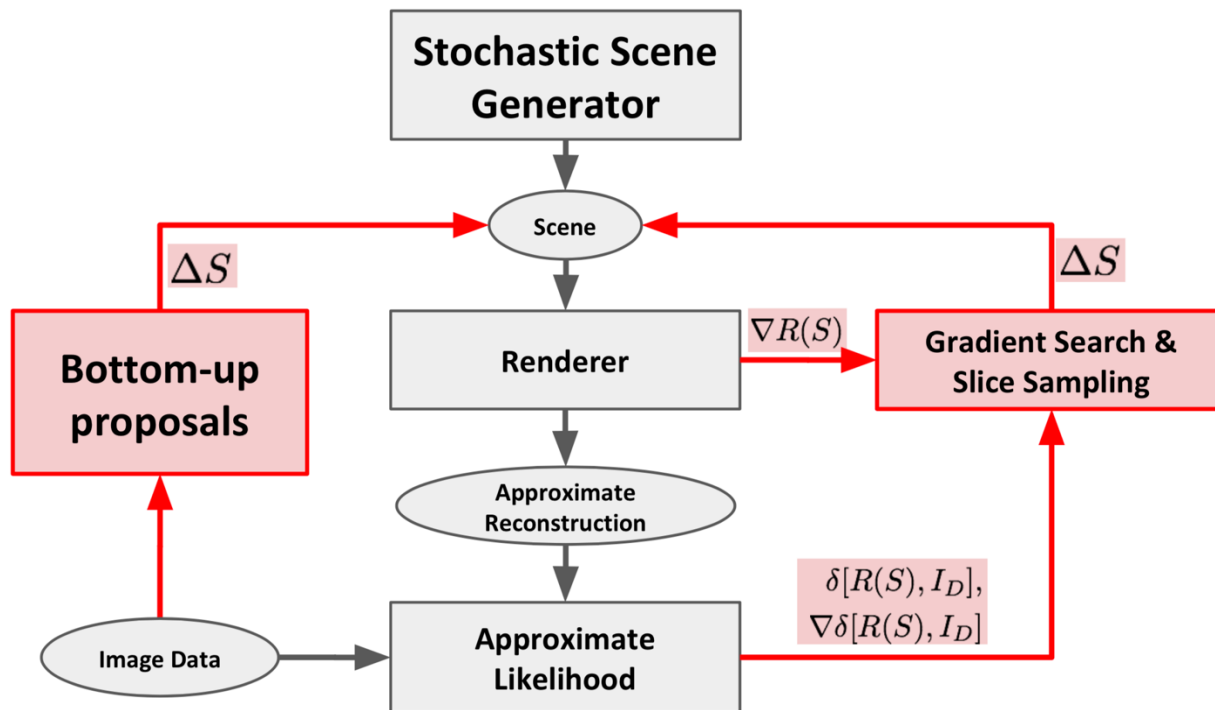


Figure 62. An overview of the inference architecture from Picture, in which learned bottom-up proposals can be combined with custom search operators. (Kulkarni et al., 2015)

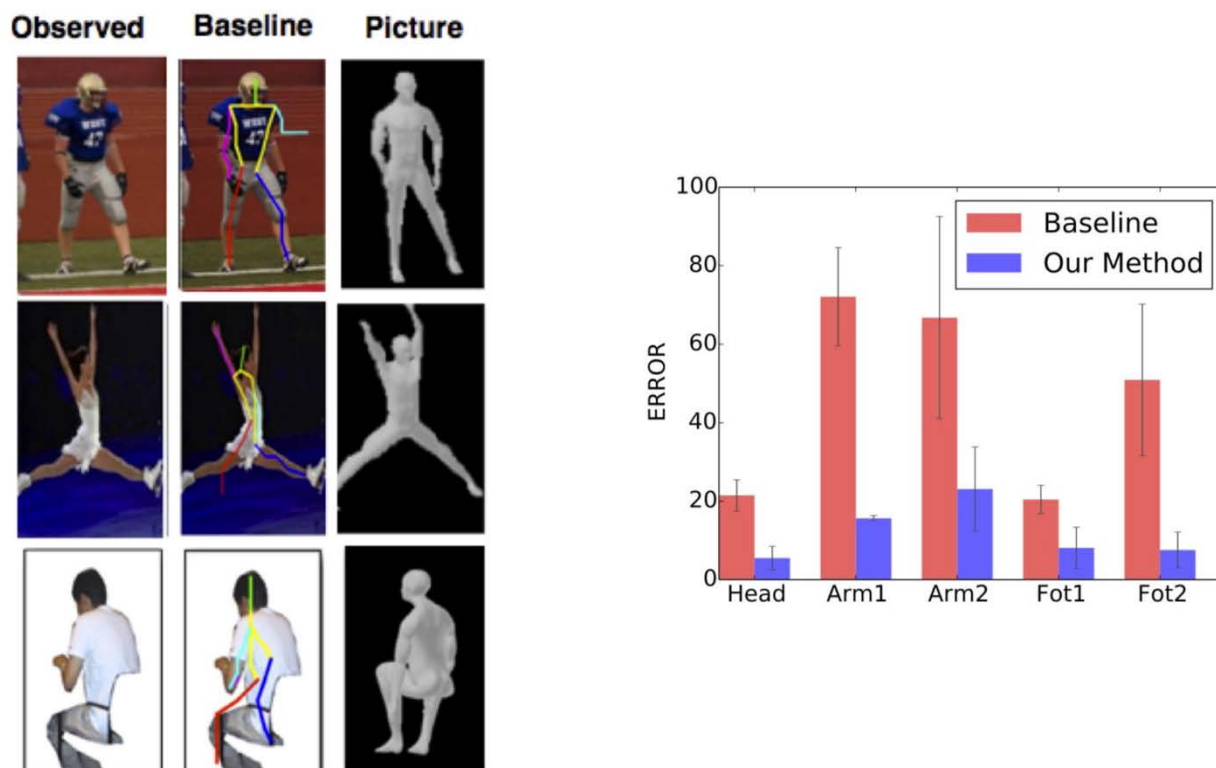
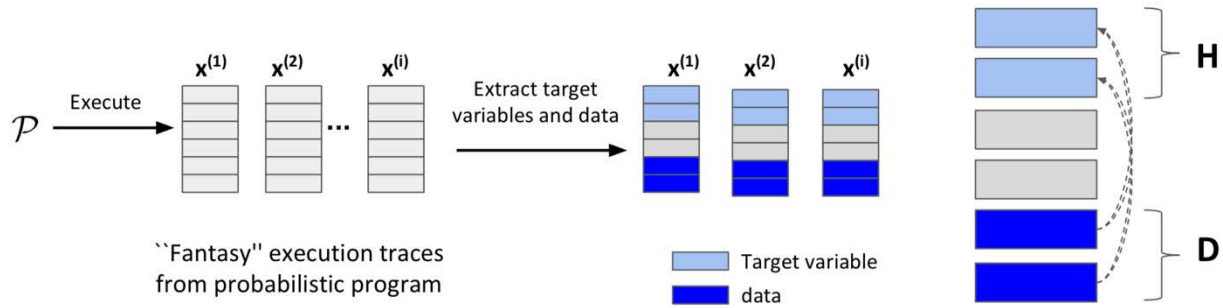


Figure 63. Quantitative and qualitative results for 3D human pose program.

We developed a Picture program for parsing 3D pose of articulated humans from single images. There has been notable work in model-based approaches for 3D human pose estimation, which served as an inspiration for the program we describe in this section. However, in contrast to Picture, existing approaches typically require custom inference strategies and significant task-specific model engineering. The probabilistic code consists of latent variables denoting bone and joints of an articulated 3D base mesh of a body. In our probabilistic code, we use an existing base mesh of a human body, defined priors over bone location and joints, and enable the armature skin-modifier API via Picture's Blender engine API. The latent scene Formula in this program can be visualized as a tree with the root node around the center of the mesh, and consists of bone location variables, bone rotation variables and camera parameters. The representation layer Formula in this program uses fine-grained image contours and the comparator is expressed as the probabilistic chamfer distance.

We evaluated our program on a dataset of humans performing a variety of poses, which was aggregated from KTH and LabelMe images with significant occlusion in the “person sitting”(around 50 total images). This dataset was chosen to highlight the distinctive value of a graphics model-based approach, emphasizing certain dimensions of task difficulty while minimizing others: While graphics simulators for articulated bodies can represent arbitrarily complex body configurations, they are limited with respect to fine-grained appearance (e.g., skin and clothing), and fast methods for fine-grained contour detection currently work well only in low clutter environments. We initially used only single-site MH proposals, although blocked proposals or HMC can somewhat accelerate inference.

We compared this approach with the discriminatively trained Deformable Parts Model (DPM) for pose estimation(referred as DPM-pose), which is notably a 2D pose model. Images with people sitting and heavy occlusion are very hard for the discriminative model to get right - mainly due to “missing” observation signal-while our model-based approach can handle these reasonably if we constrain the knee parameters to bend only in natural ways in the prior. Most of our model's failure cases, are in inferring the arm position; this is typically due to noisy and low quality feature maps around the arm area due to its small size.



Examples of "fantasy" execution traces including target variables and data

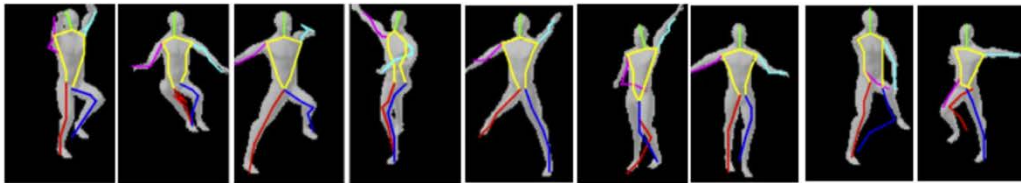


Figure 64. Bottom: Random program traces sampled from the prior during training. The colored stick figures are the results of applying DPM pose model on the hallucinated data from the program. (Kulkarni et al., 2015) This illustrates the process by which Picture programs can learn bottom-up proposals.

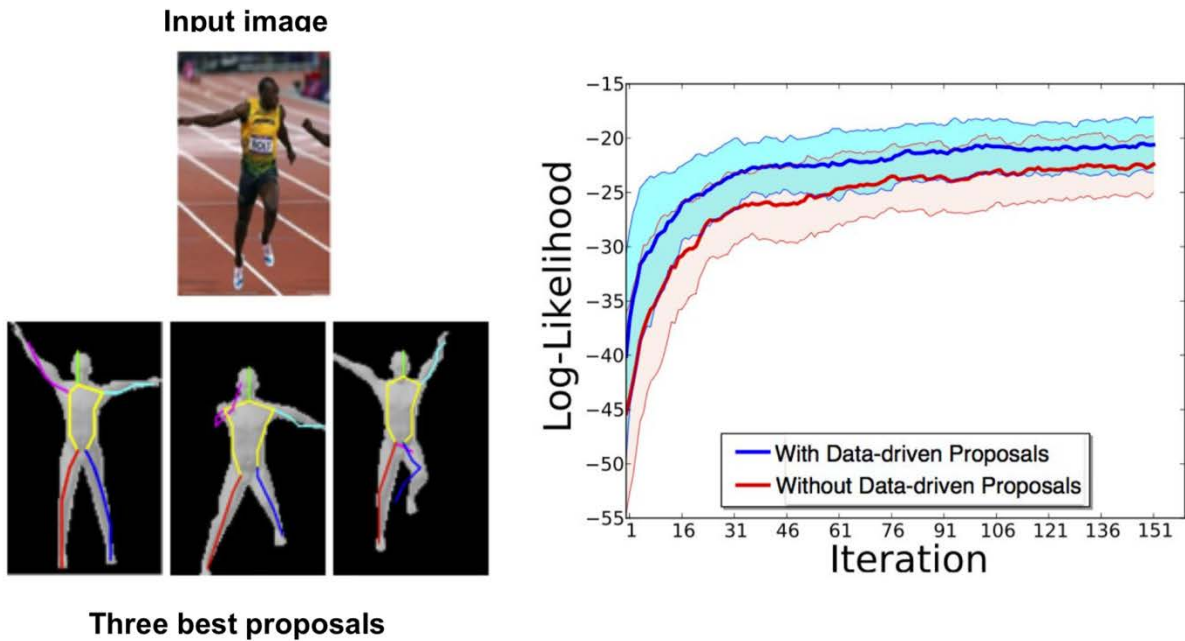


Figure 65. A detailed illustration of the utility of learned bottom-up proposals for inference in Picture programs. The top input image, as well as the three best samples drawn from the learned bottom-up proposals conditioned on the test image are semantically close to the test image and results are fine-tuned by top-down inference to close the gap. As shown on the log-1 plot, we run about 100 independent chains with and without the learned proposal. Inference with a mixture kernel of learned bottom-up proposals and single-site MH consistently outperforms baseline in terms of both speed and accuracy. (Kulkarni et al., 2015)

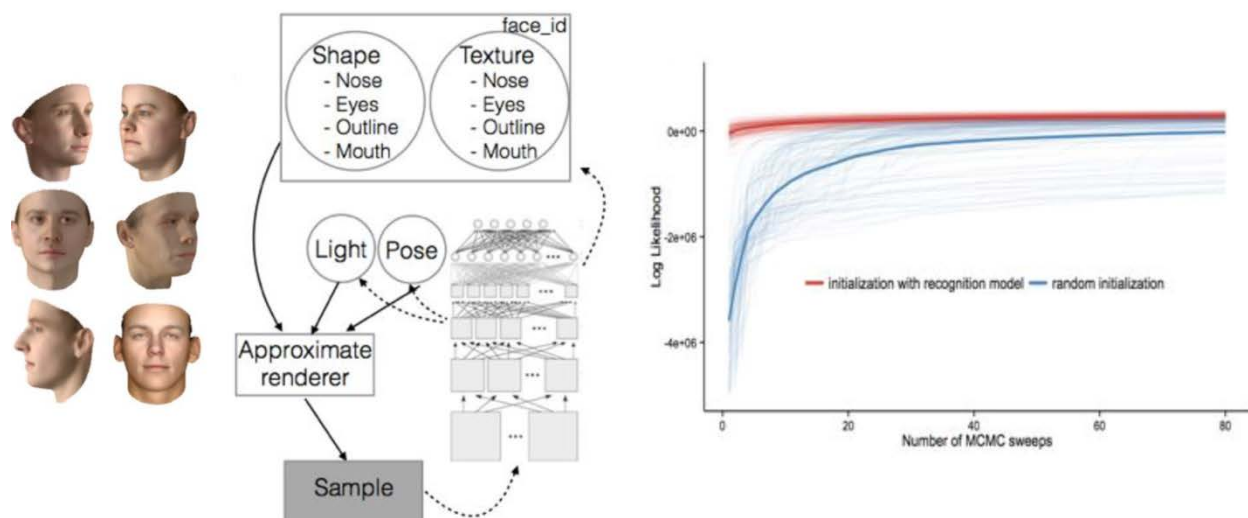


Figure 66. Left, random samples; middle, training and the use of the generative model; right, Reconstructions after MCMC iterations. (b) The average and individual log likelihood scores arising from randomly initialized 96 different chains vs. the recognition model initialized 96 chains. The recognition model initialized chains converge fast in less than 20 MCMC sweeps, and the variability across chains becomes much smaller. (Yildirim et al., 2015).

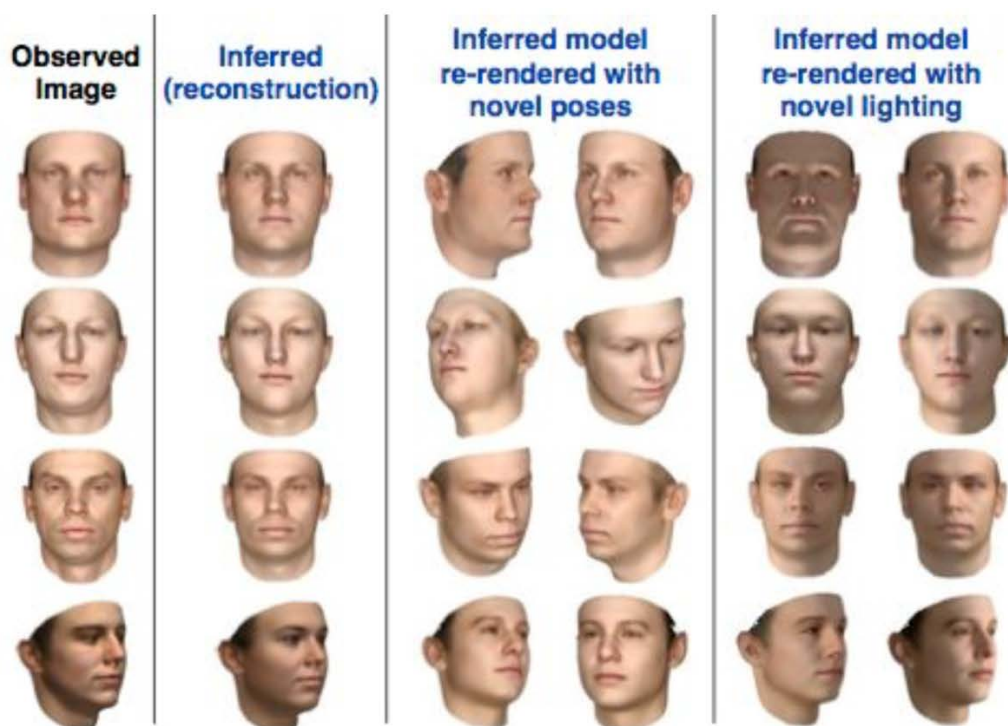


Figure 67. Picture inference on representative faces, answering the question "what does a given face probably look like when rotated or lit differently?". The first column shows the input images. The remaining columns show renderings of the inferred 3D models from a ~50-line probabilistic program written in Picture. This example shows that a short probabilistic program is applicable to non-frontal faces and provides reasonable parses, using only general-purpose inference machinery built into Picture. (Kulkarni et al., 2015)

There are many promising directions for future research in probabilistic graphics programming. Introducing a dependency tracking mechanism could let us exploit the many conditional independencies in rendering for more efficient parallel inference. Automatic particle-filter based inference schemes (Wood, van de Meent, and Mansinghka, 2014; Kulkarni, Saeedi, and Gershman, 2014) could extend the approach to image sequences. Better illumination (Zivanov et al, 2013), texture and shading models could let us work with more natural scenes. Procedural graphics techniques (Averkiou et al., 2014; Fish et al., 2013) would support far more complex object and scene models (Zhang et al., 2014; Gupta, Efros, and Hebert, 2010; del Pero et al., 2013; Hedau, Hoiem, and Forsyth, 2010). Flexible scene generator libraries will be essential in scaling up to the full range of scenes people can interpret.

We are also interested in further extending Picture by taking insights from learning based “analysis-by-synthesis” approaches such as transforming auto-encoders (Hinton, Krizhevsky, and Wang, 2011), capsule networks (Tielman 2014) and deep convolutional inverse graphics network (Kulkarni et al., 2015). These models learn an implicit graphics engine in an encoder-decoder style architecture. With probabilistic programming, the space of decoders need not be restricted to neural networks and could consist of arbitrary probabilistic graphics programs with internal parameters.

The recent renewal of interest in inverse graphics approaches to vision has motivated a number of new modeling and inference tools. Each addresses a different facet of the general problem. Earlier formulations of probabilistic graphics programming provided compositional languages for scene modeling and a flexible template for automatic inference. Differentiable renderers make it easier to fine-tune the numerical parameters of high-dimensional scene models. Data-driven proposal schemes suggest a way to rapidly identify plausible scene elements, avoiding the slow burn-in and mixing times of top-down MCMC-based inference in generative models. Deep neural networks, deformable parts models and other discriminative learning methods can be used to automatically construct good representation layers or similarity metrics for comparing hypothesized scenes to observed images. Here we have shown that by integrating all of these ideas into a single probabilistic language and inference framework, it may be feasible to begin scaling up inverse graphics to a range of real-world vision problems.

5.0 CONCLUSION

Our research in this program led to the creation of multiple open-source probabilistic programming languages and early demonstrations of their usefulness on a broad class of problems. Specifically, we have shown that probabilistic programming can (i) reduce the lines of code required to build state-of-the-art machine learning systems by ~50x; (ii) make machine learning and data science capabilities accessible to a broader class of programmers, by providing automatic model discovery mechanisms and simple, SQL-like query languages; (iii) make it possible to deploy rich generative models to solve applied problems, and thereby solve hard 3D computer vision problems with no training data, and (iv) reveal interfaces and abstractions that unify a broad set of probabilistic programming languages and enable multiple inference strategies or "solvers" to interoperate. Our research has also helped to create the technical foundations for new collaborations between the machine learning and programming language research communities.

Probabilistic programming is in its early days. There have been successful early applications in Bayesian data analysis and in probabilistic AI research. However, almost all probabilistic programs are still under 50 lines of probabilistic code, and written by a small group of people, most of whom only know how to use a single probabilistic programming language. Significant performance and usability challenges must be overcome before larger probabilistic programs will be practical to write and before broader adoption will be possible.

However, there are already new projects attempting to address these limitations. For example, Reid Hoffman (the founder of LinkedIn) recently agreed to give a \$1M gift to researchers from this PPAML team, to support the continued development of PPAML technology. A specific goal of this project is to improve usability, so that PPAML software can be applied to improve the integrity and health of the democratic process in the United States.

There are also early indications of potential use cases of PPAML technology within the defense and intelligence communities. One key focus area we believe is ready for additional research and development is "data-starved" artificial intelligence domains, such as visual scene understanding for ISR and autonomous systems applications. Another is in reducing the development cost, development time, and maintenance cost and complexity for data science applications. Both of these use cases will require additional research and development. A third is in AI-assisted scientific discovery. For example, the probabilistic programming languages developed by our team are playing an important role in DARPA's new Synergistic Discovery and Design (SD2) program, supporting AI-assisted scientific discovery and robust design for problem domains such as synthetic biology.

6.0 REFERENCES

- [1] Mansinghka, V., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint, arXiv:14040099. 2014.
- [2] Kulkarni, T.D., Kohli, P., Tenenbaum, J.B., and Mansinghka, V. Picture: A probabilistic programming language for scene perception. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR. 2015, pp. 4390-4399.
- [3] Mansinghka, V., Tibbetts, R., Baxter, J., Shafto, P., and Eaves, B. BayesDB: A probabilistic programming system for querying the probable implications of data. arXiv preprint, arXiv:1512. In review at the Journal of Machine Learning Research.
- [4] Ackerman, N., Freer, C., and Roy, D. On the computability of conditional probability. arXiv preprint, arXiv:1005:3014. 2010.
- [5] Mansinghka, V., Schaechtle, U., Handa, S., Radul, A., Chen, Y., and Martin, R. Custom Inference Strategies for Probabilistic Programming Languages. In Review, 2018.
- [6] Conway, D. (2013). "The Data Science Venn Diagram." Retrieved from <http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>, 1/5/17.
- [7] Mansinghka, V., Shafto, P., Jonas, E., Petshulat, C., Gasner, M., and Tenenbaum, J. Crosscat: A Fully Bayesian, Nonparametric Method For Analyzing Heterogeneous, High-dimensional Data. Journal of Machine Learning Research, 17. 2016.
- [8] Aly, M. "Real time detection of lane markers in urban streets". In: Intelligent Vehicles Symposium, 2008 IEEE. IEEE. 2008, pp. 7–12.
- [9] Mansinghka, V., Kulkarni, T., Perov, Y., and Tenenbaum, J. Approximate bayesian image interpretation using generative probabilistic graphics programs. In Advances in Neural Information Processing Systems, pages 1520–1528, 2013.
- [10] Gelman, A., Carlin, J., Stern, H., Dundon, D., Vehtari, A., and Rubin, D. Bayesian data analysis. Vol. 2. CRC press. 2014.
- [11] Hahnel, D., Burgard, W., Fox, D., and Thrun, S. An efficient FastSLAM algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In Intelligent Robots and Systems, 2003. Proceedings. 2003 IEEE/RSJ International Conference on, Vol. 1. IEEE, 206–211.
- [12] Liu, J. Monte Carlo strategies in scientific computing. 2008. Springer Science & Business Media.

- [13] Murphy, K. Machine learning: a probabilistic perspective. MIT press. 2012.
- [14] Russell, S., and Norvig, P. Artificial Intelligence: A Modern Approach (2nd ed.). Pearson Education. 2003.
- [15] Thrun, S., Burgard, W., and Fox, D. Probabilistic robotics. MIT press. 2005.
- [16] Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. An introduction to MCMC for machine learning. *Machine learning* 50, 1-2 (2003), 5–43.
- [17] Huang, D., Tristant, J. and Morrisett, G. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *Proceedings of the 38th ACM SIG-PLAN Conference on Programming Language Design and Implementation*. ACM, 111–125, 2017.
- [18] Lindsten, F., Jordan, M., and Schoen, T. Particle gibbs with ancestor sampling. *Journal of Machine Learning Research* 15, 1 (2014), 2145–2184.
- [19] Murray, L. Bayesian state-space modelling on high-performance hardware using LibBi. *arXiv preprint arXiv:1306.3277* (2013).
- [20] Neal, R. Slice sampling. *Annals of statistics* (2003), 705–741.
- [21] Ranganath, R., Gerrish, S., and Blei, D. Black box variational inference. In *Artificial Intelligence and Statistics*. 814–822, 2014.
- [22] Wingate, D. and Weber, T. Automated variational inference in probabilistic programming. *arXiv*, preprint *arXiv:1301.1299*, 2013.
- [23] Carpenter, B., Gelman, A., Hoffman, M., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. Stan: A probabilistic programming language. *Journal of Statistical Software* 20 (2016), 1–37. 2016.
- [24] Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., and Tenenbaum, J. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*. 2008.
- [25] Goodman, N. and Stuhlmüller, A. The Design and Implementation of Probabilistic Programming Languages. <http://dippl.org>. (2014). Accessed: 2017-11-15. 2014.

- [26] Gordon, A., Graepel, T., Rolland, N., Russo, C., Borgstrom, J., and Guiver, J. Tabular: a schema-driven probabilistic programming language. In ACM SIGPLAN Notices, Vol. 49. ACM, 321–334. 2014.
- [27] Gordon, A., Henzinger, T., Nori, A., and Rajamani, S. Probabilistic programming. In Proceedings of the on Future of Software Engineering. ACM, 167–181. 2014.
- [28] Uber AI Labs. Pyro, a deep probabilistic programming Language. (2017). <https://eng.uber.com/pyro/>
- [29] Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., and Kolobov, A. BLOG: Probabilistic models with unknown objects. Statistical relational learning (2007), 373.
- [30] Tolpin, D., van de Meent, J., and Wood, F. Probabilistic programming in Anglican. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Springer, 308–311, 2015.
- [31] Tristan, J., Huang, D., Tassarotti, J., Pocock, A., Green, S., and Steele, G. Augur: Data-parallel probabilistic modeling. In Advances in Neural Information Processing Systems. 2600–2608. 2014.
- [32] Mansinghka, V., Selsam, D., and Perov, Y. Venture: a higher-order probabilistic programming platform with programmable inference. arXiv preprint arXiv:1404.0099, 2014.
- [33] Radul, A. and Mansinghka, V. Metaprob: a simple, extensible language for probabilistic programming and meta-programming. Chapter in forthcoming volume from MIT Press. 2018.
- [34] Saad, F. and Mansinghka, V. A Bayesian Nonparametric Method for Clustering Imputation, and Forecasting in Multivariate Time Series. arXiv preprint arXiv:1710.06900, 2017.
- [35] Saad, F., and Mansinghka, V. Probabilistic Data Analysis with Probabilistic Programming. arXiv 1608.05347, 2016. In review at the Journal of Machine Learning Research.
- [36] Codd, E. A relational model of data for large shared data banks. Communications of the ACM, 13(6):377–387, 1970
- [37] Automated Statistician. “About.” James Robert Lloyd. Web. Accessed January 10, 2018.
- [38] Duvenaud, D., Lloyd, J., Grosse, R., Tenenbaum, J., and Ghahramani, Z. Structure discovery in nonparametric regression through compositional kernel search. In Proceedings of the International Conference on Machine Learning (ICML), pages 1166–1174, 2013.

- [39] Schaechtle, U., Zinberg, B., Radul, A., Stathis, K., and Mansinghka, V. Probabilistic programming with Gaussian process memoization. arXiv preprint, arXiv:1512.05665, 2015.
- [40] Tong, A. and Choi, J. Automatic generation of probabilistic programming from time series data. arXiv preprint, arXiv:1607.00710, 2016.
- [41] Stan Development Team. Stan Modeling Language Users Guide and Reference Manual, Version 2.0, 2013. URL <http://mc-stan.org/>.
- [42] Schaechtle, U., Saad, F., Radul, A., and Mansinghka, V. Time Series Structure Discovery via Probabilistic Program Synthesis. arXiv preprint, arXiv:1611.07051. 2017.
- [43] Grosse, R., Salakhutdinov, R., Freeman, W., and Tenenbaum, J. Exploiting compositionality to explore a large space of model structures. In Proceedings of the Twenty-Eighth Conference Annual Conference on Uncertainty in Artificial Intelligence, pages 306–31, Corvallis, Oregon, 2012. AUAI Press.
- [44] Mansinghka, V., Kemp, C., Griffiths, T., and Tenenbaum, J. Structured priors for structure learning. In Proceedings of the Twenty-Second Conference Annual Conference on Uncertainty in Artificial Intelligence, pages 324–33, 2006.
- [45] Felzenszwalk, P., Girshick, D., McAllester, D., and Ramanan, D. Object detection with discriminatively trained partbased models. PAMI, 2010.
- [46] Krizhevsky, A., Sutskever, I., and Hinton, G. Imagenet classification with deep convolutional neural networks. In NIPS, pages 1106–1114, 2012.
- [47] LeCun, Y. and Bengio, Y. Convolutional networks for images, speech, and time series. The Handbook of Brain Theory and Neural Networks, 3361, 1995.
- [48] Lowe, D. Distinctive image features from scale-invariant keypoints. IJCV, 2004.
- [49] Grenander, U. General pattern theory-A mathematical study of regular structures. Clarendon Press, 1993.
- [50] Grenander, U., Chow, Y., and Keenan, D. Hands: A pattern theoretic study of biological shapes. Springer-Verlag New York, Inc., 1991.
- [51] Zhao, Y. and Zhu, S. Image parsing via stochastic scene grammar. In NIPS, 2011.
- [52] Del Pero, L., Bowdish, J., Kermgard, B., Hartley, E., and Barnard, K. Understanding bayesian rooms using composite 3d object models. In Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on, pages 153–160. IEEE, 2013.

- [53] Jampani, V., Nowozin, D., Loper, M., and Gehler, P. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. arXiv preprint arXiv:1402.0859, 2014.
- [54] Loper, M. and Black, M. Opendr: An approximate differentiable renderer. In ECCV 2014. 2014.
- [55] Hinton, G., Osindero, S. and Teh, Y. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- [56] Jin, Y. and Geman, S. Context and hierarchy in a probabilistic image model. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2145–2152. IEEE, 2006.
- [57] Yildirim, Y., Kulkarni, T., Freiwald, W., and Tenenbaum, J. Efficient analysis-by-synthesis in vision: A computational framework, behavioral tests, and modeling neuronal representations. *COGSCI*, 2015.
- [58] Wood, F., van de Meent, J., and Mansinghka, J. A new approach to probabilistic programming inference. In *AISTATS*, 2014.
- [59] Kulkarni, T., Saeedi, A., and Gershman, S. Variational particle approximations. arXiv preprint arXiv:1402.5715, 2014.
- [60] Zivanov, J., Forster, A., Schonborn, S., and Vetter, T.. Human face shape analysis under spherical harmonics illumination considering self occlusion. In *Biometrics (ICB), 2013 International Conference on*, pages 1–8. IEEE, 2013.
- [61] Averkiou, M., Kim, V., Zheng, Y., and Mitra, N. Shapessynth: Parameterizing model collections for coupled shape exploration and synthesis. In *Computer Graphics Forum*, volume 33, pages 125–134. Wiley Online Library, 2014.
- [62] Fish, N., Averkiou, M., Van Kaick, O., Sorkine-Hornung, O., Cohen-Or, D. and Mitra, N. Meta-representation of shape families. In *Computer Graphics Forum*, volume 32, pages 189–200, 2013.
- [63] Zhang, Y., Song, S., Tan, P., and Xiao, J. Panocontext: A wholeroom 3d context model for panoramic scene understanding. In *Computer Vision–ECCV 2014*, pages 668–686. Springer, 2014.
- [64] Gupta, A., Efros, A. and Hebert, M. Blocks world revisited: Image understanding using qualitative geometry and mechanics. In *Computer Vision–ECCV 2010*, pages 482–496. Springer, 2010.

- [65] Hedau, V., Hoiem, D., and Forsyth, D. Thinking inside the box: Using appearance models and context based on room geometry. In *Computer Vision–ECCV 2010*, pages 224–237. Springer, 2010.
- [66] Hinton, G., Krizhevsky, A., and Wang, S. Transforming auto-encoders. In *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 44–51. Springer, 2011.
- [67] Tieleman, T. Optimizing Neural Networks that Generate Images. PhD thesis, University of Toronto, 2014.
- [68] Kulkarni, T., Whitney, W., Kohli, P., and Tenenbaum, J. Deep convolutional inverse graphics network. *arXiv preprint arXiv:1503.03167*, 2015.

LIST OF ACRONYMS

API	Application Programming Interface
ast	Abstract Syntax Tree
BQL	Bayesian Query Language
CGPM	Compositional Generative Population Model
DPM	Deformable Parts Model
MCMC	Markov Chain Monte Carlo
MML	Meta-modeling Language
NIPS	Neural Information Processing Systems
PPAML	Probabilistic Programming for Advancing Machine Learning
SD2	Synergistic Discovery and Design
TA	Technical Area